UNIT-1

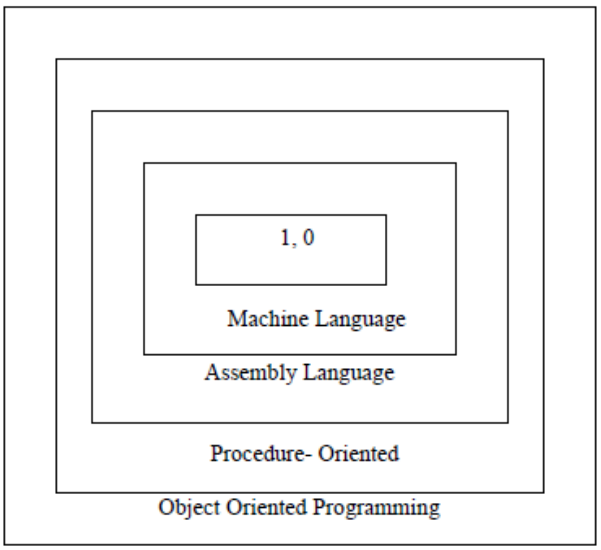
C++ Programming Elements, Classes and Objects, Constructors and Destructors, Static Class Members, Dynamic Memory Allocation (new and delete), Passing Parameter Methods, Inline Functions, Friend Functions.

Software Crisis:

- Developments in software technology continue to be dynamic.
- New tools and techniques are announced in quick succession.
- This has forced the software engineers and industry to continuously look for new approaches to software design and development.
- These rapid advances appear to have created a situation of crisis within the industry
- The following issued need to be addressed to face the crisis:
  - ✓ How to represent real-life entities of problems in system design?
  - ✓ How to design system with open interfaces?
  - ✓ How to improve the quality of software?
  - ✓ How to manage time schedules?
  - ✓ How to ensure reusability and extensibility of modules?
  - ✓ How to develop modules that are tolerant of any changes in future?
  - ✓ How to improve software productivity and decrease software cost?

Layers of Computer Software:

Ernest Tello, A well-known writer in the field of artificial intelligence, compared the evolution of software technology to the growth of the tree. Like a tree, the software evolution has had distinct phases "layers" of growth. These layers were building up one by one over the last five decades as shown in fig. 1.1, with each layer representing and improvement over the previous one. However, the analogy fails if we consider the life of these layers.



Procedure/ Structure Oriented Programming:

Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as procedure oriented programming (POP).

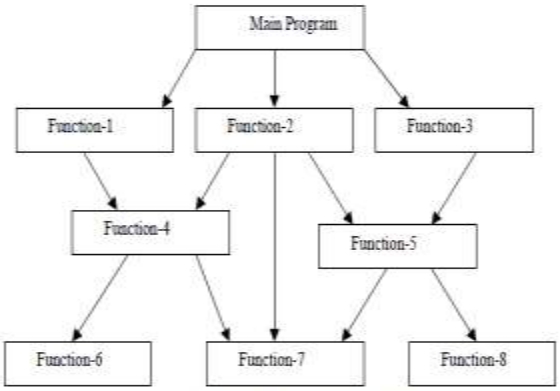
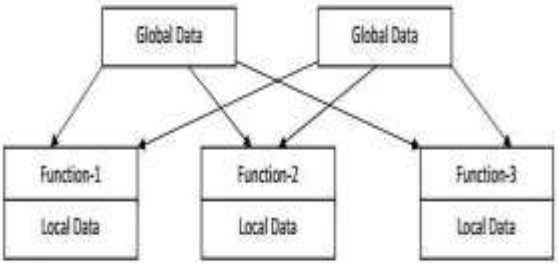


Fig. 1.2 Typical structure of procedural oriented programs

In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on functions. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem.

Another serious drawback with the procedural approach is that we do not model real world problems very well. This is because functions are action-oriented and do not really corresponding to the element of the problem.

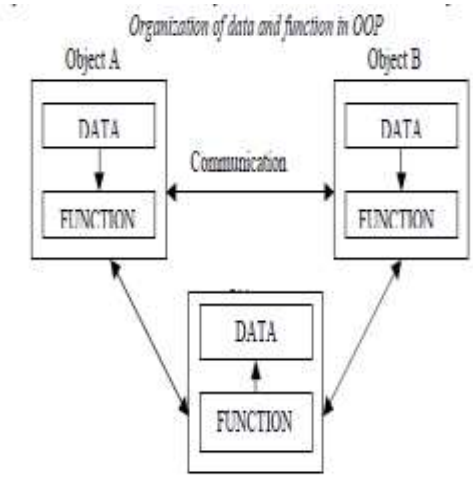


Some Characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

Object Oriented Design:

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The organization of data and function in object-oriented programs is shown in fig.1.3. The data of an object can be accessed only by the function associated with that



object. However, function of one object can access the function of other objects.

Some of the features of object oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are ties together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.

Comparison of POP & OOP:

Type	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called functions.	In OOP, program is divided into parts called objects.
Importance	In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.
Approach	POP follows Top Down approach.	OOP follows Bottom Up approach.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is less secure.	OOP provides Data Hiding so provides more security.

Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	C, VB, FORTRAN, Pascal.	C++, JAVA, VB.NET, C#.NET.

Basic Concepts of Object Oriented Programming:

It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

- ✓ Objects
- ✓ Classes
- ✓ Data abstraction and encapsulation
- ✓ Inheritance
- ✓ Polymorphism
- ✓ Dynamic binding
- ✓ Message passing

Objects:

Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in term of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in c.

Classes:

We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects similar types.

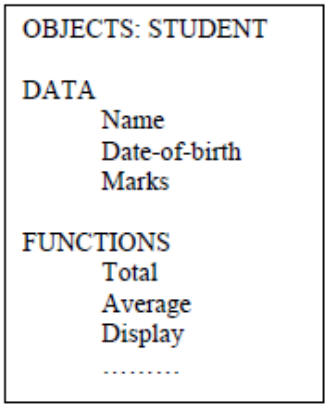


Fig. 1.5 representing an object

For examples: Mango, Apple and orange members of class fruit.

Classes are user-defined that types and behave like the built-in types of a programming language. The syntax used to create an object is not different then the syntax used to create an integer object in C.

If fruit has been defines as a class, then the statement Fruit Mango; will create an object **mango** belonging to the class **fruit.**

Data Abstraction and Encapsulation:

The wrapping up of data and function into a single unit (called class) is known as *encapsulation.* Data and encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called *data hiding or information hiding*.

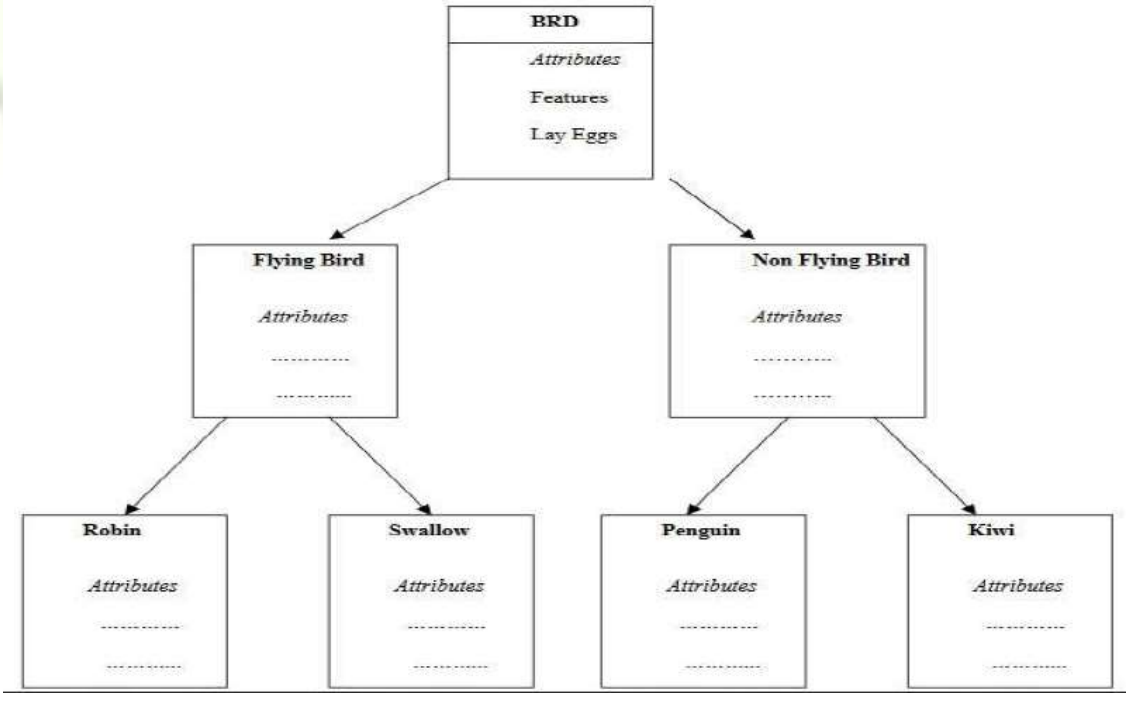
Abstraction refers to the act of representing essential features without including the background details or explanation. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes. They encapsulate all the essential properties of the object that are to be created.

The attributes are sometime called *data members* because they hold information. The functions that operate on these data are sometimes called *methods or member function*.

Inheritance:

Inheritance is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of *hierarchical classification.* For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'. The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in figure.

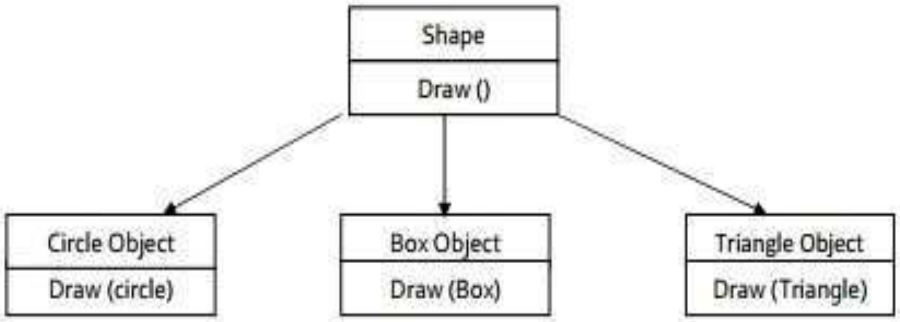
In OOP, the concept of inheritance provides the idea of *reusability*. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes.



Polymorphism:

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than on form. An operation may exhibit different behavior is different

instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as *operator overloading*.



The figure illustrates that a single function name can be used to handle different number and different types of argument. This is something similar to a particular word having several different meanings depending upon the context. Using a single function name to perform different type of task is known as *function overloading.*

Dynamic Binding:

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.
Consider the procedure "draw" in fig. 1.7. by inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

Message Passing:

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, involves the following basic steps:

1. Creating classes that define object and their behavior,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

Benefits of OOP:

OOP offers several benefits to both the program designer and the user. Object-Orientation contributes to the solution of many problems associated with the development and quality of

software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.
The principal advantages are:

- Through inheritance, we can eliminate redundant code extend the use of existing Classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure program that cannot be invaded by code in other parts of a programs.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map object in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more detail of a model can implemental form.
- Object-oriented system can be easily upgraded from small to large system.
- Message passing techniques for communication between objects makes to interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

Application of OOP:

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as window. Hundreds of windowing systems have been developed, using the OOP techniques.
Real-business system are often much more complex and contain many more objects with complicated attributes and method. OOP is useful in these types of application because it can simplify a complex problem. The promising areas of application of OOP include:

- Real-time system
- Simulation and modeling
- Object-oriented data bases
- Hypertext, Hypermedia, and expert Ext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

History of C++:

- The C++ programming language was developed in 1983 by **Bjarne Stroustrup** at AT&T (American Telephonic & Telegraphic) Bell Laboratories, New Jersey. It was mainly influenced by the languages Simula-67, C and ALGOL 68.
- Before 1983, the language was called as **"C with Classes"**, as it was simply an extension to C language. Hence, all the concepts of C were also applicable in C++.
- It was develop for adding a feature of **OOP (Object Oriented Programming)** in C without significantly changing the C component.

- C++ programming is "relative" (called a superset) of C, it means any valid C program is also a valid C++ program.

Difference b/w C and C++:

No.	C	C++
1)	C follows the **procedural style programming.**	C++ is multi-paradigm. It supports both **procedural and object oriented.**
2)	Data is less secured in C.	In C++, you can use modifiers for class members to make it inaccessible for outside users.
3)	C follows the **top-down approach.**	C++ follows the **bottom-up approach.**
4)	C does not support function overloading.	C++ supports function overloading.
5)	In C, you can't use functions in structure.	In C++, you can use functions in structure.
6)	C does not support reference variables.	C++ supports reference variables.
7)	In C, **scanf() and printf()** are mainly used for input/output.	C++ mainly uses stream **cin and cout** to perform input and output operations.
8)	Operator overloading is not possible in C.	Operator overloading is possible in C++.
9)	C programs are divided into **procedures and modules**	C++ programs are divided into **functions and classes.**
10)	C does not provide the feature of namespace.	C++ supports the feature of namespace.
11)	Exception handling is not easy in C. It has to perform using other functions.	C++ provides exception handling using Try and Catch block.
12)	C does not support the inheritance.	C++ supports inheritance.

C++ Features:

C++ is object oriented programming language. It provides a lot of **features** that are given below.

1. **Simple**: C++ is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2. **Machine Independent or Portable:** Unlike assembly language, c programs can be executed in many machines with little bit or no change. But it is not platform-independent.
3. **Mid-level programming language:** C++ is also used to do low level programming. It is used to develop system applications such as kernel, driver etc. It also supports the feature of high level language. That is why it is known as mid-level language.
4. **Structured programming language:** C++ is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.
5. **Rich Library:** C++ provides a lot of inbuilt functions that makes the development fast.
6. **Memory Management:** It supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory at any time by calling the free() function.
7. **Fast Speed:** The compilation and execution time of C++ language is fast.
8. **Pointers:** C++ provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array etc.
9. **Recursion:** In C++, we can call the function within the function. It provides code reusability for every function.
10. **Extensible:** C++ language is extensible because it can easily adopt new features.
11. **Object Oriented:** C++ is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
12. **Compiler based:** C++ is a compiler based programming language, it means without compilation no C++ program can be executed. First we need to compile our program using compiler and then we can execute our program.

C++ Basics:

Basic Input/ Output:

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as **output operation.**

If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as **input operation.**

The common header files used in C++ programming are:

Header File	Function and Description
<iostream>	It is used to define the **cout, cin and cerr** objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.
<iomanip>	It is used to declare services useful for performing formatted I/O, such as **setprecision and setw.**
<fstream>	It is used to declare services for user-controlled file processing.

The **cout** is a predefined object of **ostream** class. It is connected with the standard output device, which is usually a display screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console.

The **cin** is a predefined object of **istream** class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

The **endl** is a predefined object of **ostream** class. It is used to insert a new line characters and flushes the stream.

Variables:

Variable is an entity which can change its value during the execution of the program. The length of variable name is infinite in C++. It allows any alphabets and numeric values and a special symbol _.

Valid Variables: C, Amount, sub1, Total_Marks, number1, number2, _num, average;
Invalid Variables: if, true, percentage%, $price, 1class, total marks;

All variable must be declared before they are used in the program.

Data Types:

Types	Data Types
Basic Data Type	int, char, float, double, etc
Derived Data Type	array, pointer, etc
Enumeration Data Type	enum
User Defined Data Type	structure

Keywords:

A keyword is a reserved word. You cannot use it as a variable name, constant name etc. In C++, we have **32 Keywords** which are also available in C language and in addition to that it has **30 keywords. Totally C++ has 62 Keywords.**

C++ Operators:

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc.

There are following types of operators to perform different types of operations in C language.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operator
- Ternary or Conditional Operator
- Increment/ Decrement Operators / Unary Operator
- Assignment Operator

Control Statements:

There are various types of control statements in C++.

Conditional/Selection control statements: if, if-else, nested if, else-if ladder, switch

Looping/ Iteration Statements: while, do-while, for

Unconditional/ Jump Statements: break, continue, return, goto

Sample Program:

```
#include <iostream>
using namespace std;
int main( ) {
  int age;
   cout << "Enter your age: ";
   cin >> age;
   cout << "Your age is: " << age << endl;
}
```

1. Program feature:
Like C, the C++ program is a collection of function. The above example contain only one function **main().** As usual execution begins at main(). Every C++ program must have a **main()**. C++ is a free form language. With a few exception, the compiler ignore carriage return and white spaces. Like C, the C++ statements terminate with semicolons.

2. Comments:
C++ introduces a new comment symbol // (double slash). Comment start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line, and whatever follows till the end of the line is ignored. Note that there is no closing symbol.
The double slash comment is basically a single line comment. Multiline comments can be written as follows:

```
// This is an example of
// C++ program to illustrate
// some of its features
```

The C comment symbols /*,*/ are still valid and are more suitable for multiline comments. The following comment is allowed:

```
/* This is an example of
C++ program to illustrate
some of its features
*/
```

3. Output operator:
The only statement in program 1.10.1 is an output statement. The statement Cout<<"C++ is better than C.";
Causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, cout and <<. The identifier cout(pronounced as C out) is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices. The operator << is called the insertion or put to operator.

4. The iostream File:
We have used the following #include directive in the program:

```
#include <iostream>
```

The #include directive instructs the compiler to include the contents of the file enclosed within angular brackets into the source file. The header file **iostream.h** should be included at the beginning of all programs that use input/output statements.

5. Namespace:
Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifier defined in the **namespace** scope we must include the using directive, like

```
Using namespace std;
```

6. Return Type of main():
In C++, main () returns an integer value to the operating system. Therefore, every main () in C++ should end with a return (0) statement; otherwise a warning an error might occur. Since main () returns an integer type for main () is explicitly specified as **int.** Note that the default return type for all function in C++ is **int.** The following main without type and return will run with a warning:

```
main ()
{
……..
……
}
```

Functions:

A complex problem is often easier to solve by dividing it into several smaller parts, each of which can be solved by it. This is called ***structured*** programming. These parts are sometimes made into ***functions*** in C++.

To perform any task, we can create function. A function can be called many times. It provides modularity and code reusability. C++ allows the use of both internal (user-defined) and external functions.

There are two types of functions in C programming:

1. Library Functions: are the functions which are declared in the C++ header files such as ceil(x), cos(x), exp(x), etc.

2. User-defined functions: are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.

Syntax:

```
return_type      function_name(data_type parameter...)
        {
        //code to be executed
        }
```

Passing Parameter Methods:

a. **Call by Value: -** In this method the values of the actual parameters (appearing in the function call) are copied into the formal parameters (appearing in the function definition), i.e., the function creates its own copy of argument values and operates on them. The following program illustrates this concept:

```
#include<iostream>
using namespace std;
```

```
void swap(int x, int y)
{
 int swap;
 swap = x;
 x = y;
 y = swap;
}
int main()
{
 int a=500, b=100;
cout<<"Before Swapping"<<endl;
 cout<<"Value of a is: "<<a<<endl;
 cout<<"Value of b is: "<<b<<endl;

 swap(a, b);  // passing value to function
cout<<"After Swapping"<<endl;
cout<<"Value of a is: "<<a<<endl;
 cout<<"Value of b is: "<<b<<endl;
 return 0;
}
```

b. Call by Reference: - A reference provides an alias – an alternate name – for the variable, i.e., the same variable's value can be used by two different names: the original name and the alias name.

In call by reference method, a reference to the actual arguments(s) in the calling program is passed (only variables). So the called function does not create its own copy of original value(s) but works with the original value(s) with different name. Any change in the original data in the called function gets reflected back to the calling function.

It is useful when you want to change the original variables in the calling function by the called function.

```
#include<iostream>
using namespace std;
void swap(int &x, int &y)
{
 int swap;
 swap = x;
 x = y;
 y = swap;
}
int main()
{
 int a=500, b=100;
cout<<"Before Swapping"<<endl;
 cout<<"Value of a is: "<<a<<endl;
```

```
 cout<<"Value of b is: "<<b<<endl;

 swap(a, b);  // passing value to function
cout<<"After Swapping"<<endl;
cout<<"Value of a is: "<<a<<endl;
 cout<<"Value of b is: "<<b<<endl;
 return 0;
}
```

Recursion:

When function is called within the same function, it is known as **recursion**.

Example:

- Factorial of a given number
  - *n! = n * (n – 1) * (n – 2) * ... * 1*
- Recursive relationship (*n! = n * (n – 1)!*)
  - *5! = 5 * 4!*
  - *4! = 4 * 3! ...*
- Base case (*1! = 0! = 1*)

```
#include<iostream>
using namespace std;
int main()
{
int factorial(int);
int fact,value;
cout<<"Enter any number: ";
cin>>value;
fact=factorial(value);
cout<<"Factorial of a number is: "<<fact<<endl;
return 0;
}
int factorial(int n)
{
if(n<0)
return(-1); /*Wrong value*/
if(n==0)
return(1);  /*Terminating condition*/
else
{
return(n*factorial(n-1));
}
}
```

Inline Functions:

These are the functions designed to speed up program execution. An inline function is expanded (i.e. the function code is replaced when a call to the inline function is made) in the line where it is invoked. You are familiar with the fact that in case of normal functions, the compiler has to jump to another location for the execution of the function and then the control is returned back to the instruction immediately after the function call statement. So execution time taken is more in case of normal functions. There is a memory penalty in the case of an inline function.

In summary:

***The inline** function:*

- Reduce function-call overhead
- Asks the compiler to copy code into program instead of using a function call
- Compiler can ignore inline
- Should be used with small, often-used functions

Example:

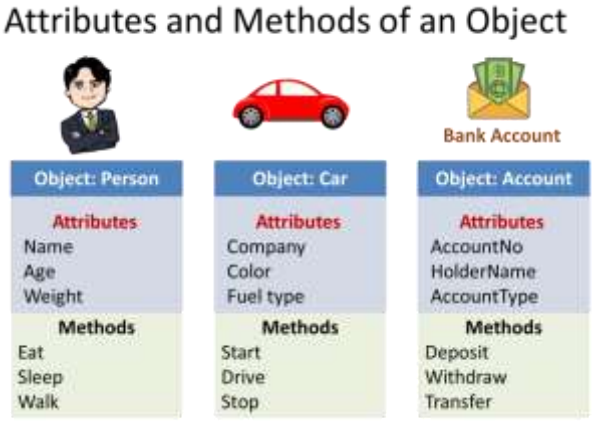
```
inline double cube( const double s )
{
        return s * s * s;
}
```

Default Arguments:

- If function parameter omitted, gets default value
  - Can be constants, global variables, or function calls
  - If not enough parameters specified, rightmost go to their defaults
- Set defaults in function prototype
  - int fun1(int x = 1, int y = 2, int z = 3);

Classes & Objects:

Object is a real-world entity which contains the state and behavior. Object is a collection of data and methods.



Example:

Physical Objects: Pen, Bike, Student, Laptop, Projector, etc.

Logical Objects: Bank accounts, Results, Stock exchange, weather reports etc.

Class is a **blueprint** or **template** that describes the object. Class specifies the attributes and methods of objects. With the help of classes we can create objects. Objects are the **instances** of a class. We can create n no. of objects with the help of a class.

Syntax:

class <classname>
 	
{
 	
// data members and member functions
 	
} object1;
	class <classname>
	
	{
	
	// data members and member functions
	
	};

To create objects: **className objectVariableName;**

Example:

class car
{
private:
int price;
float mileage;
public:
void start();
void drive();
};
	int main()
	{
	car c1;
	c1.start();
	}
	// where c1 is an **object name** of a class,
	We can access the members of a class by using **dot '.'** operator.

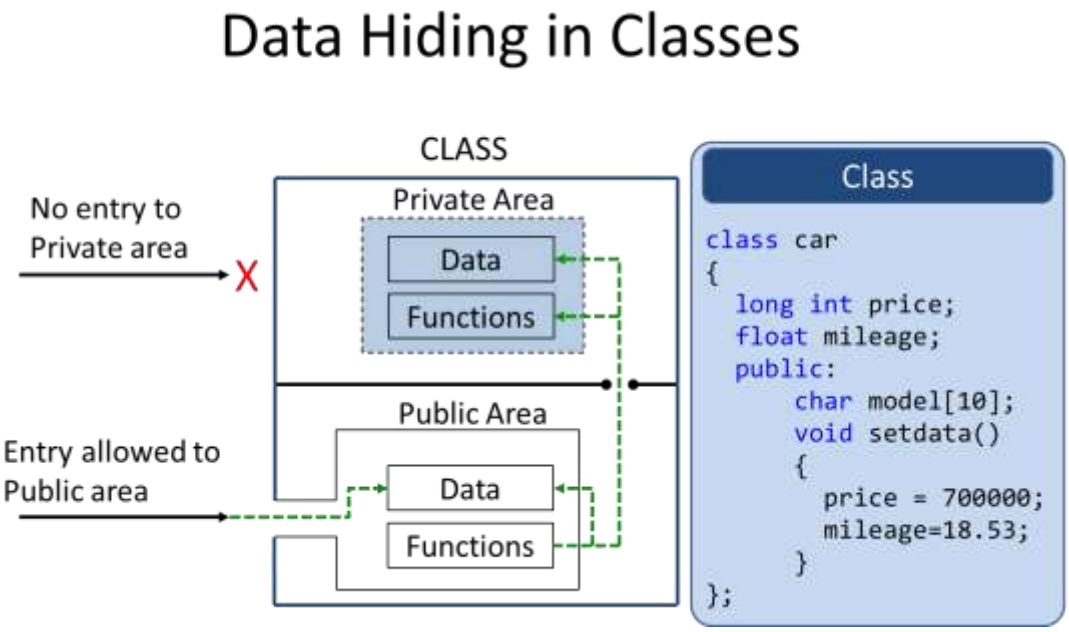
Access Modifiers:

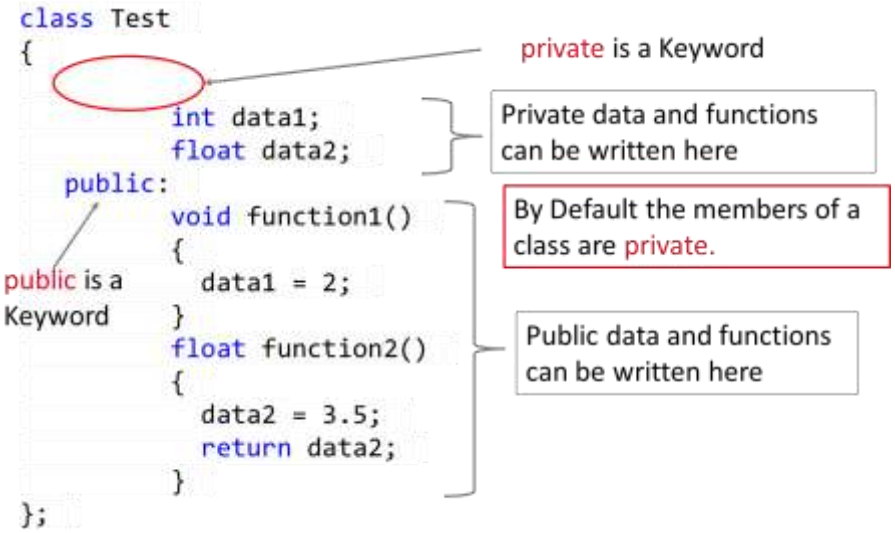
Private:

- Class **Private** Members of the class can be accessed within the class and from member functions of the class.
- They cannot be accessed outside the class or from other programs, not even from inherited class.
- If you try to access private data from outside of the class, compiler throws error.
- This feature in OOP is known as **Data hiding / Encapsulation**.
- By default the members in a class are **Private**, if any other access modifier is not specified.

Public:

- The **public** keyword makes data and functions public.
- Public members of the class are accessible by any program from anywhere.
- Class members that allow manipulating or accessing the class data are made public.
- The **public** members of a class can be accessed outside the class using the **object name** and dot operator '.'





Example Programs on Classes and Objects:

Program-1:

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int mark;
    float spi;
  public:
    void SetData()
    {
      mark = 270;
      spi = 6.5;
    }
    void DisplayData()
    {
      cout << "Mark= "<<mark<<endl;
      cout << "spi= "<<spi;
    }
} ;
int main()
{
  Test o1;
  o1.SetData();
  o1.DisplayData();
  return 0;
}
```

Program-2:

```
#include<iostream>
using namespace std;
class Car
{
 private:
   char company[20];
   int top_speed;
 public:
   void SetData(){
      cout<<"Enter Company:";
      cin>>company;
      cout<<"Enter top speed:";
      cin>>top_speed;
```

```
    }
    void DisplayData()
    {
      cout << "\nCompany:"<<company;
      cout << "\tTop Speed:"<<top_speed;
    }
} ;
int main()
{
  Car o1;
  o1.SetData();
  o1.DisplayData();
  return 0;
}
```

Program-3:

```
#include<iostream>
using namespace std;
class Employee
{
  private:
    char name[10];
    int salary, age;

  public:
    void SetData()
    {
          cout << " Enter name salary and age of an employee : " ;
          cin>>name>>salary>>age;
    }
    void DisplayData()
    {
      cout << "Name= "<<name<<endl;
      cout << "salary= "<<salary<<endl;
      cout << "age= "<<age;
    }
} ;
int main()
{
  Employee o1;
  o1.SetData();
  o1.DisplayData();
  return 0;
}
```

Function definition outside the class:

If you want to define the member function definition of a class outside to that class by using **scope resolution operator (::).**

Syntax:

Return-type **class-name ::** function-name(arguments)

{

Function body;

}

Note The membership label **class-name::** tells the compiler that the function belongs to a particular class.

Program-4:

```
#include<iostream>
using namespace std;
class car
{
 private:
   float mileage;
 public:
  float updatemileage();
   void setdata();

};

float car :: updatemileage()
{
 return mileage+2;
}

void car :: setdata()
{
  mileage = 18.5;
}

int main()
{
```

```
  car c1;
  c1.setdata();
  c1.updatemileage();
}
```

Program-5:

```
#include<iostream>
using namespace std;
class Test
{
  private:
   int mark;
   float spi;
  public:
   void SetData(int,float);
   void DisplayData();
};
void Test :: SetData(int i,float j){
   mark = i;
   spi = j;
}
void Test :: DisplayData()
{
   cout << "Mark= "<<mark;
   cout << "\nspi= "<<spi;
}
int main()
 {
   Test o1;
   o1.SetData(70,6.5);
   o1.DisplayData();
   return 0;
 }
```

Member Functions with Arguments

Program-6:

```
#include<iostream>
using namespace std;
class Time
{
   private :
     int hour, minute, second;
   public :
```

```
    void setTime(int h, int m, int s);
    void print();
};
void Time::setTime(int h, int m, int s)
{
 hour=h;
 minute=m;
 second=s;
}
void Time::print()
{
 cout<<"hours=\n"<<hour;
 cout<<"minutes=\n"<<minute;
 cout<<"seconds=\n"<<second;
}
int main()
{
 int h,m,s;
 Time t1;
 cout<<"Enter hours="; cin>>h;
 cout<<"Enter minutes="; cin>>m;
 cout<<"Enter seconds="; cin>>s;

 t1.setTime(h,m,s);
 t1.print();
 return 0;
}
```

Program-7:

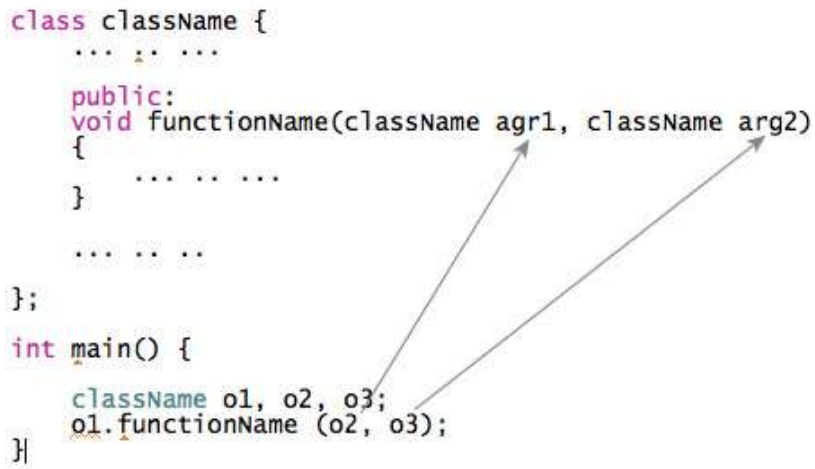
```
#include<iostream>
using namespace std;
class Rectangle
{
  int width, height;
  public:
    void set_values (int,int);
    int area(){
      return width*height;
    }
};
void Rectangle::set_values (int x, int y){
  width = x;  height = y;
}
int main(){
  Rectangle rect;
```

```
  rect.set_values(3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

Program-8:

```
#include<iostream>
using namespace std;
class Employee{
  private :
    int age; int salary;
  public :
    void setData(int , int);
    void displaydata();
};
void Employee::setData(int x, int y){
  age=x;
  salary=y;
}
void Employee::displaydata(){
  cout<<"age="<<age<<endl;
  cout<<"salary="<<salary<<endl;
}
int main(){
  Employee yash,raj;
  yash.setData(23,1500);
  yash.displaydata();

  raj.setData(27,1800);
  raj. displaydata();
  return 0;
}
```

Object as Function arguments:



Program-9:

```
#include<iostream>
using namespace std;
class Time
{
    int hour, minute, second;
  public :
    void getTime(){
      cout<<"\nEnter hours:";cin>>hour;
      cout<<"Enter Minutes:";cin>>minute;
      cout<<"Enter Seconds:";cin>>second;
    }
    void printTime(){
      cout<<"\nhour:"<<hour;
      cout<<"\tminute:"<<minute;
      cout<<"\tsecond:"<<second;
    }
    void addTime(Time x, Time y){
      hour = x.hour + y.hour;
      minute = x.minute + y.minute;
      second = x.second + y.second;
    }
};
int main()
{
  Time t1,t2,t3;

  t1.getTime();
  t1.printTime();
```

```
  t2.getTime();
  t2.printTime();

  t3.addTime(t1,t2);
  cout<<"\n after adding two objects";
  t3.printTime();

  return 0;
}
```

Program-10:

```
#include<iostream>
using namespace std;
class Complex
{
 private:
   int real,imag;
 public:
   void readData()
   {
     cout<<"Enter real and imaginary number:";
     cin>>real>> imag;
   }
   void addComplexNumbers(Complex comp1, Complex comp2)
   {
     real=comp1.real+comp2.real;
     imag=comp1.imag+comp2.imag;
   }
   void displaySum()
   {
     cout << "Sum = " << real<< "+" << imag << "i";
   }
};
int main()
{
   Complex c1,c2,c3;
   c1.readData();
   c2.readData();
   c3.addComplexNumbers(c1, c2);
   c3.displaySum();
}
```

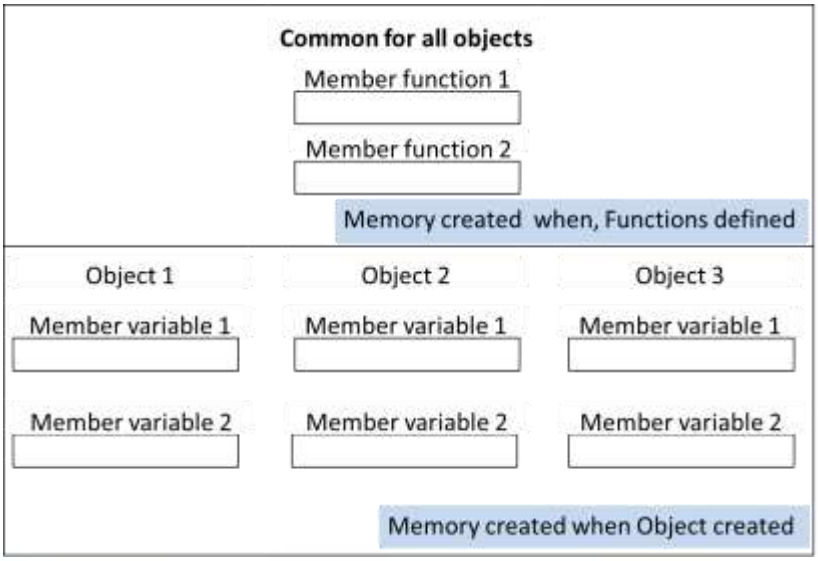
Returning Object:

Program-11:

```
#include<iostream>
using namespace std;
class Time{
    int hour, minute, second;
  public :
    void getTime(){
     cout<<"\nEnter hours:";cin>>hour;
     cout<<"Enter Minutes:";cin>>minute;
    }
    void printTime(){
     cout<<"\nhour:"<<hour;
     cout<<"\tminute:"<<minute;
    }
    Time addTime(Time t1, Time t2){
     Time t4;
     t4.hour = t1.hour + t2.hour;
     t4.minute = t1.minute + t2.minute;
     return t4;
    }
};
int main()
{
 Time t1,t2,t3,ans;

 t1.getTime();
 t1.printTime();

 t2.getTime();
 t2.printTime();

 ans=t3.addTime(t1,t2);
 cout<<"\nafter adding two objects";
 ans.printTime();

 return 0;
}
```

Memory Allocation of Objects:

- The **member functions** are created and placed in the memory space **only once** at the time they are defined as part of a class specification.
- No separate space is allocated for member functions when the **objects** are created.
- Only space for **member variable** is allocated separately for each **object** because, the member variables will hold different data values for different objects.




```
class Account
{
  int Account_no,Balance;
  char Account_type[10];
 public:
  void setdata(int an,char at[],int bal)
  {
       Account_no = an;
       Account_type = at;
       Balance = bal;
  }
};
```

```
int main(){
 Account A1,A2,A3;
 A1.setdata(101,"Current",3400);
 A2.setdata(102,"Saving",150);
 A3.setdata(103,"Current",7900);
 return 0;
}
```

Object	A1
Account No	
Account Type	
Balance	

Object	A2
Account No	
Account Type	
Balance	

Object	A3
Account No	
Account Type	
Balance	

Static Data Members:

A static data member is useful, when all objects of the same class must **share common information**. In other words Data members of the class which are shared by all objects are known as **static** data members.

- Just write static keyword prefix to regular variable
- **Only one copy** of a static variable is maintained by the class and it is common for all objects.
- **Static members** are **declared** inside the class and **defined** outside the class.
- It is initialized to **zero** when the first object of its class is created.
- You **cannot** initialize a static member variable inside the class declaration.
- It is visible only within the class but its lifetime is the entire program.
- **Static members** are generally used to maintain values common to the entire class.

Program-12:

```
#include<iostream>
using namespace std;
class item
{
  int number;
  static int count;// static variable declaration
  public:
  void getcount(){
        number=0;
        number++;
    count++;
   cout<<"\nvalue of count: "<<count;
    cout<<"\nvalue of number: "<<number;
  }
};
int item :: count;   // static variable definition
int main()
{
 item a,b,c;
 a.getcount();
 b.getcount();
 c.getcount();
 return 0;
}
```

Static Member Functions:

- **Static member functions** can access only static members of the class.
- **Static member functions** can be invoked using **class name**, not object.
- There cannot be static and non-static version of the same function.
- They cannot be **virtual**.

- They cannot be declared as **constant** or **volatile**.
- A static member function does not have **this pointer**.

Program-13:

```
#include<iostream>
using namespace std;
class item
{
  int number;
  static int count;// static variable declaration
  public:
  void getdata(int a){
    number = a;
    count++;
  }
  static void getcount(){
    cout<<"value of count: "<<count;
  }
};
int item :: count; // static variable definition
int main()
{
 item a,b,c;
 a.getdata(100);
 item::getcount();
 b.getdata(200);
 item::getcount();
 c.getdata(300);
 item::getcount();
 return 0;
}
```

Friend Function:

- In C++ a **Friend Function** that is a "friend" of a given class is allowed **access to private and protected data** in that class.
- A friend function is a function which is declared using **friend** keyword.
- **Friend function** can be declared either in public or private part of the class.
- It is not a member of the class so it **cannot be called using the object**.
- Usually, it has the **objects as arguments**.

Program-14:

```
class numbers {
  int num1, num2;
  public:
```

```
   void setdata(int a, int b);
   friend int add(numbers N);
};
void numbers :: setdata(int a, int b){
  num1=a;
  num2=b;
}
int add(numbers N){
  return (N.num1+N.num2);
}
int main()
{
  numbers N1;
  N1.setdata(10,20);
  cout<<"Sum = "<<add(N1);
  return 0;
}
```

Program-15:

```
class  base
{
    int val1,val2;
   public:
    void get(){
       cout<<"Enter two values:";
       cin>>val1>>val2;
    }
    friend float mean(base ob);
};
float mean(base ob){
   return float(ob.val1+ob.val2)/2;
}
int main()
{
    base obj;
    obj.get();
    cout<<"\n Mean value is : "<<mean(obj);
}
```

Program-16: Friend function to another class

```
class ABC {
 private:
  int numA;
 public:
```

```
  void setdata(){
   numA=10;
  }
 friend int add(ABC, XYZ);
};
class XYZ {
 private:
  int numB;
 public:
  void setdata(){
    numB=25;
  }
 friend int add(ABC , XYZ);
};

int add(ABC objA, XYZ objB){
  return (objA.numA + objB.numB);
}
int main(){
  ABC objA;   XYZ objB;
  objA.setdata();  objB.setdata();
  cout<<"Sum: "<< add(objA, objB);
}
```

Use of friend function:

- It is possible to grant a nonmember function access to the private members of a class by using a friend function.
- It can be used to overload binary operators.

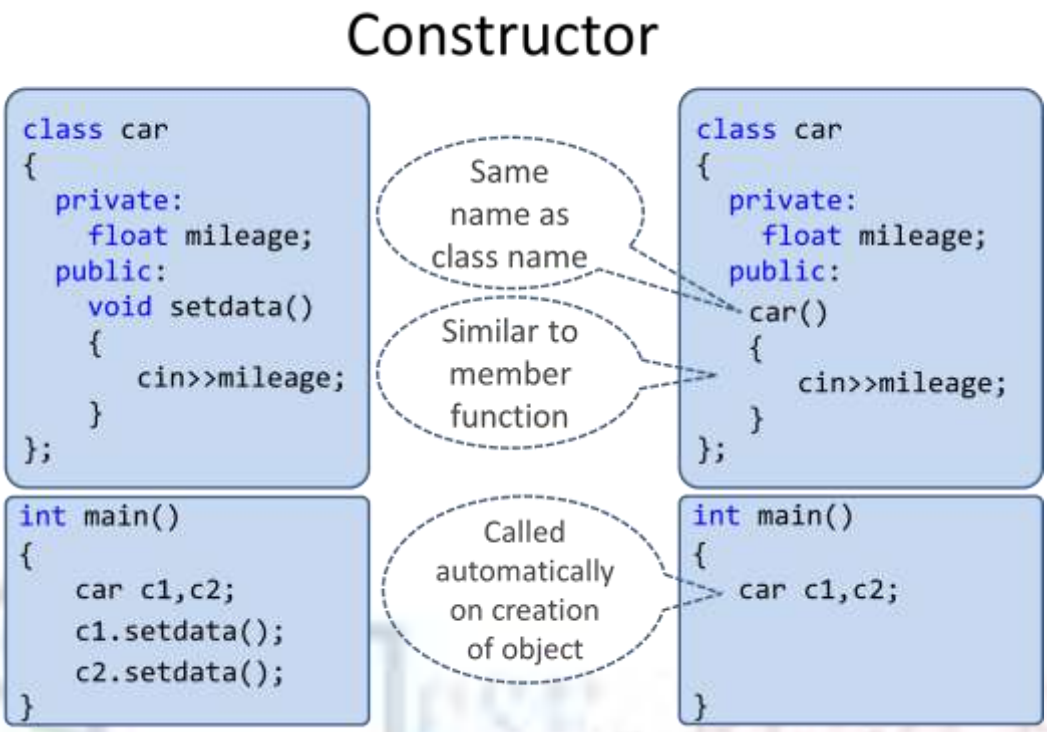
Constructors:

A constructor is a block of code which is,

- ✓ similar to member function
- ✓ has same name as class name
- ✓ **Do not have return types** and they cannot return values, not even void.
- ✓ called automatically when object of class created

A **constructor** is used to initialize the objects of class as soon as the object is created. **Constructor** should be declared in public section because private constructor cannot be invoked outside the class so they are useless.

- ➢ Constructors **cannot be inherited**, even though a derived class can call the base class constructor.
- ➢ Constructors **cannot be virtual**.
- ➢ They make implicit calls to the operators **new** and **delete** when memory allocation is required.



Types of Constructors:

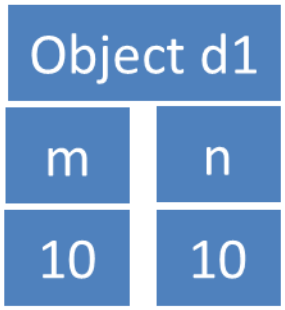
1) Default constructor
2) Parameterized constructor
3) Copy constructor

1) Default Constructor

- Default constructor is the one which invokes by default when object of the class is created.
- It is generally used to initialize the default value of the data members.
- It is also called no argument constructor.

```
class demo{
 int m,n;
public:
 demo()
 {
  m=n=10;
 }
};
```

```
int main()
{
 demo d1;
}
```



Program-17:

```
class Area
{
 private:
  int length, breadth;
 public:
  Area(){
  length=5;
  breadth=2;
  }
  void Calculate(){
   cout<<"\narea="<<length * breadth;
  }
};
int main(){
 Area A1;
 A1.Calculate();
 Area A2;
 A2.Calculate();
 return 0;
}
```

2) Parameterized Constructor

- Constructors that can take arguments are called **parameterized constructors.**
- Sometimes it is necessary to initialize the various data elements of different objects with different values when they are created.
- We can achieve this objective by passing arguments to the constructor function when the objects are created.

Program-18:

```
class demo
{
  int m,n;
  public:
  demo(int x,int y){ //Parameterized Constructor
    m=x;
    n=y;
    cout<<"Constructor Called";
  }
};
int main()
{
 demo d1(5,6);
}
```

3) Copy Constructor:

- A **copy constructor** is used to declare and initialize an object from another object using an object as argument.

For example:

demo(demo &d); //declaration

demo d2(d1); //copy object

OR demo d2=d1; //copy object

- Constructor which accepts a reference to its own class as a parameter is called **copy constructor**.

Program-19:

```
class demo
{
  int m, n;
  public:
  demo(int x,int y){
   m=x;
   n=y;
   cout<<"Parameterized Constructor";
  }
  demo(demo &x){
   m = x.m;
   n = x.n;
   cout<<"Copy Constructor";
  }
};
int main()
{
  demo obj1(5,6);
  demo obj2(obj1);
  demo obj2 = obj1;
}
```

Program-20:

```
class rectangle{
  int length, width;
  public:
  rectangle(){ // Default constructor
    length=0;
    width=0;
  }
  rectangle(int x, int y){// Parameterized constructor
    length = x;
    width = y;
  }
  rectangle(rectangle &_r){ // Copy constructor
    length = _r.length;
    width = _r.width;
  }
```

```
};

int main()
{
  rectangle r1; // Invokes default constructor
  rectangle r2(10,20); // Invokes parameterized constructor
  rectangle r3(r2); // Invokes copy constructor
}
```

Destructor:

- **Destructor** is used to destroy the objects that have been created by a constructor.
- The syntax for **destructor** is same as that for the constructor,
  - ✓ the class name is used for the name of destructor,
  - ✓ with a **tilde (~)** sign as prefix to it.
- Never takes any argument nor it returns any value nor does it have return type.
- It is invoked automatically by the complier upon exit from the program.
- It should be declared in the public section.

Program-21:

```
class rectangle
{
 int length, width;
 public:
 rectangle(){ //Constructor
  length=0;
  width=0;
  cout<<"Constructor Called";
 }
 ~rectangle() //Destructor
  {
  cout<<"Destructor Called";
  }
// other functions for reading, writing and processing can be written here
};
int main()
{
  rectangle x;
// default constructor is called
}
```

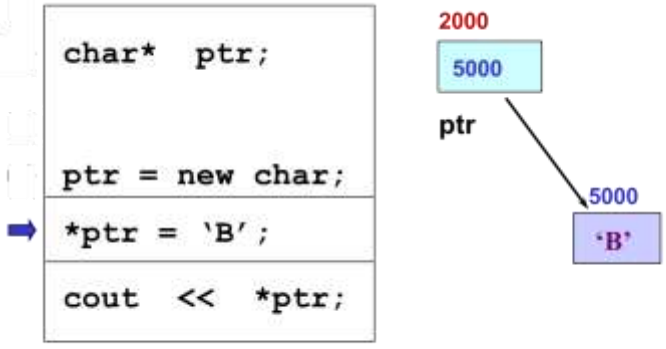
Dynamic Memory Allocation:

- *In C*, functions such as malloc() are used to dynamically allocate memory from the **Heap**.
- *In C++*, this is accomplished using the **new** and **delete** operators
- **new** is used to allocate memory during execution time
  - ✓ returns a pointer to the address where the object is to be stored
  - ✓ always returns a pointer to the type that follows the **new**

new DataType

new DataType [IntExpression]

- If memory is available, in an area called the heap (or free store) new allocates the requested object or array, and returns a pointer to (address of) the memory allocated.
- Otherwise, program terminates with error message.
- The dynamically allocated object exists until the delete operator destroys it.



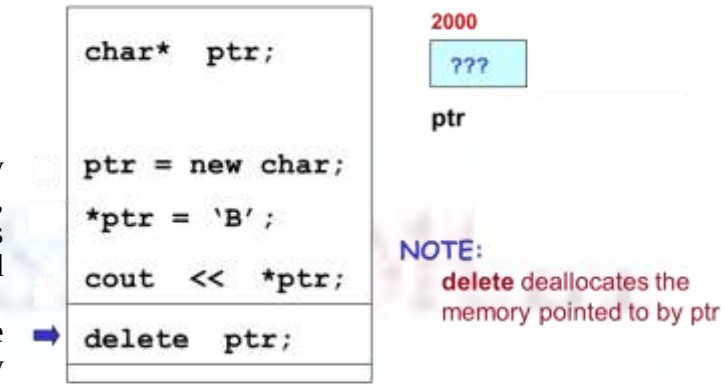
NOTE: Dynamic data has no variable name

Operator delete Syntax

delete Pointer

delete [ ] Pointer

- The object or array currently pointed to by Pointer is deallocated, and the value of Pointer is undefined. The memory is returned to the free store.
- Square brackets are used with delete to deallocate a dynamically allocated array.



UNIT-2

Inheritance in C++- Inheritance Types, Base class Access Control, Examples of Inheritance, Virtual Base Classes and Abstract Classes, Constructors in Derived Classes, Polymorphism - Types of Polymorphism, Function Overloading and Operator Overloading - Unary and Binary Operator Overloading.

Inheritance:

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

The class which inherits the members of another class is called **derived class** and the class whose members are inherited is called **base class**. The derived class is the specialized class for the base class.

- **Inheritance** is mechanism by which one class acquires the properties (data and operations) of another class
- Base Class (or superclass): the class being inherited from
- Derived Class (or subclass): the class that inherits

Advantages of inheritance:

- When a class inherits from another class, there are three benefits:
  - ✓ You can ***reuse*** the methods and data of the existing class.
  - ✓ You can ***extend*** the existing class by adding new data and new methods.
  - ✓ You can ***modify*** the existing class by overloading its methods with your own implementations.

NOTE:

- **A class inherits the *behavior* of another class and enhances it in some way.**
- **Inheritance *does not* mean inheriting access to another class' private members.**

Rules for building a class hierarchy:

- Derived classes are **special cases** of base classes.
- A derived class **can also serve** as a base class for new classes.
- There is no limit on the **depth of inheritance** allowed in C++ (as far as it is within the limits of your compiler)
- It is possible for a class to be a base class for **more than one** derived class.

A Derived class is defined as the class derived from the base class.
The Syntax of Derived class:

```
class derived_class_name : visibility-mode base_class_name
{
   // body of the derived class.
}
```

Where,
derived_class_name: It is the name of the derived class.
visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.
base_class_name: It is the name of the base class.

- When the base class is privately inherited by the derived class, public members of the base class become the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- ✓ In C++, the default mode of visibility is **private**.
- ✓ The private members of the base class are **never inherited**.

Access Rights of Derived Classes

- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.
- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Base class visibility	Derived class visibility		
	Public	**Private**	**Protected**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

- The type of inheritance defines the minimum access level for the members of derived class that are inherited from the base class
- With public inheritance, the derived class follow the same access permission as in the base class
- With protected inheritance, only the public members inherited from the base class can be accessed in the derived class as protected members
- With private inheritance, none of the members of base class is accessible by the derived class

Constructor Rules for Derived Classes:

The default constructor and the destructor of the base class are always called when a new object of a derived class is created or destroyed.

```
class A {
  public:
        A ( )
          {cout<< "A:default"<<endl;}
        A (int a)
          {cout<<"A:parameter"<<endl;}
}
class B : public A
{
  public:
        B (int a)
           {cout<<"B"<<endl;}
}
int main()
{
        B b1(100);
        return 0;
}
```

Output:
A:default
B

You can also specify a constructor of the base class other than the default constructor

```
DerivedClassCon ( derivedClass args ) : BaseClassCon ( baseClass args )
{
        DerivedClass constructor body
 }
```

```
class A {
  public:
        A ( )
          {cout<< "A:default"<<endl;}
```

```
        A (int a)
          {cout<<"A:parameter"<<endl;}
}
class B : public A
{
  public:
        B (int a): A(a)
           {cout<<"B"<<endl;}
}
int main()
{
        B b1(100);
        return 0;
}
```

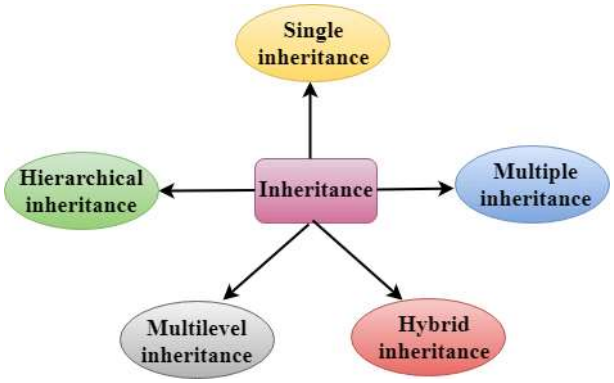
Output:
A:parameter
B

Note:

- You cannot override a base class constructor with a derived class constructor (rather, the derived class constructor calls the base class constructor first)
- All base class destructors should be declared virtual
- Virtual destructors are called in reverse order from the constructors for derived class objects

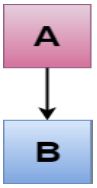
Types of Inheritance:

- Single inheritance (only one super class)
- Multiple inheritance (several super classes)
- Hierarchical inheritance (one super class, many sub classes)
- Multi-Level inheritance (derived from a derived class)
- Hybrid inheritance (more than two types)
  - Multi-path inheritance (inheritance of some properties from two sources).



Single inheritance:

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

Program-1:

```
#include <iostream>
using namespace std;
class A
{
   int a = 4;
   int b = 5;
   public:
   int mul()
   {
      int c = a*b;
      return c;
   }
};

class B : private A
{
   public:
   void display()
   {
      int result = mul();
      std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
   }
};
int main()
{
  B b;
  b.display();

   return 0;
}
```

Output:

Multiplication of a and b is : 20

Program-2:

```
#include <iostream>
using namespace std;
class base    //single base class
{
  public:
    int x;
  void getdata()
  {
    cout << "Enter the value of x = ";
    cin >> x;
  }
```

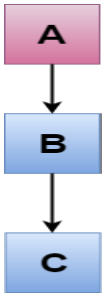
```
};
class derive : public base    //single derived class
{
  private:
   int y;
  public:
  void readdata()
  {
    cout << "Enter the value of y = ";
    cin >> y;
  }
  void product()
  {
    cout << "Product = " << x * y;
  }
};
int main()
{
   derive a;     //object of derived class
   a.getdata();
   a.readdata();
   a.product();
   return 0;
}
```

Output:

Enter the value of x = 3
Enter the value of y = 4
Product = 12

Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.



Program-3:

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
```

```
    cout<<"Eating..."<<endl;
  }
  };
  class Dog: public Animal
  {
     public:
   void bark(){
   cout<<"Barking..."<<endl;
    }
  };
  class BabyDog: public Dog
  {
     public:
   void weep() {
   cout<<"Weeping...";
    }
  };
int main(void) {
   BabyDog d1;
   d1.eat();
   d1.bark();
    d1.weep();
    return 0;
}
```

Output:

Eating...
Barking...
Weeping...

Program-4:

```
#include <iostream>
using namespace std;
class base //single base class
{
        public:
        int x;
        void getdata()
        {
        cout << "Enter value of x= ";
        cin >> x;
        }
};
```

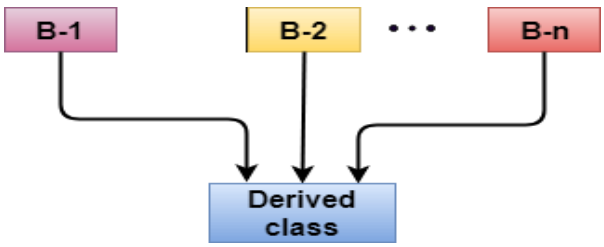
```
class derive1 : public base
{       public:
        int y;
        void readdata()
        {
           cout << "\nEnter value of y= ";
           cin >> y;
        }
};
class derive2 : public derive1
{
        private:
        int z;
        public:
        void indata()
        {
        cout << "\nEnter value of z= ";
        cin >> z;
        }
        void product()
        {
           cout << "\nProduct= " << x * y * z;
        }
};
int main()
{
    derive2 a;
    a.getdata();
    a.readdata();
    a.indata();
    a.product();
    return 0;
}
```

Output:

Enter value of x= 2
Enter value of y= 3
Enter value of z= 3
Product= 18

Multiple- Inheritance:

Multiple-inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Program-4:

```
#include <iostream>
using namespace std;
```

```
class A
{
   protected:
    int a;
   public:
   void get_a(int n)
   {
      a = n;
   }
};

class B
{
   protected:
   int b;
   public:
   void get_b(int n)
   {
      b = n;
   }
};
class C : public A,public B
{
  public:
   void display()
   {
      std::cout << "The value of a is : " <<a<< std::endl;
      std::cout << "The value of b is : " <<b<< std::endl;
      cout<<"Addition of a and b is : "<<a+b;
   }
};
int main()
{
  C c;
  c.get_a(10);
  c.get_b(20);
  c.display();

   return 0;
}
```

Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

Program-5:

```
#include <iostream>

using namespace std;

class Mammal {

  public:
```

```
    Mammal()
    {
      cout << "Mammals can give direct birth." << endl;
    }
};
class WingedAnimal {
  public:
    WingedAnimal()
    {
      cout << "Winged animal can flap." << endl;
    }
};
class Bat: public Mammal, public WingedAnimal {
};
int main()
{
    Bat b1;
    return 0;
}
```

Output:

Mammals can give direct birth.
Winged animal can flap.

Ambiguity Resolution in Inheritance:

Ambiguity can be occurred in using the multiple-inheritance when a function with the same name occurs in more than one base class.

Program-6:

```
#include <iostream>
using namespace std;
class A
{
   public:
   void display()
   {
      std::cout << "Class A" << std::endl;
   }
};
class B
```

```
{
   public:
   void display()
   {
      std::cout << "Class B" << std::endl;
   }
};
class C : public A, public B
{
   void view()
   {
      display();
   }
};
int main()
{
   C c;
   c.display();
   return 0;
}
```

Output:
error: reference to 'display' is ambiguous
display();

The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B
{
   void view()
   {
      A :: display();      // Calling the display() function of class A.
      B :: display();      // Calling the display() function of class B.

   }
};
```

An ambiguity can also occur in single inheritance.

Consider the following situation:

```
class A
{
  public:
void display()
{
  cout<<"Class A";
}
} ;
```

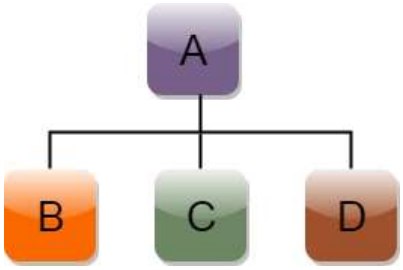
```
class B
{
  public:
 void display()
{
 cout<<"Class B";
}
} ;
```

In the above case, the function of the derived class overrides the method of the base class. Therefore, call to the display() function will simply call the function defined in the derived class. If we want to invoke the base class function, we can use the class resolution operator.

```
int main()
{
    B b;
   b.display();              // Calling the display() function of B class.
   b.B :: display();       // Calling the display() function defined in B class.
}
```

Hierarchical Inheritance:

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Program-7:

```
#include <iostream>
using namespace std;
class A //single base class
{
   public:
     int x, y;
     void getdata()
     {
        cout << "\nEnter value of x and y:\n";
        cin >> x >> y;
     }
};
class B : public A //B is derived from class base
{
   public:
     void product()
     {
        cout << "\nProduct= " << x * y;
     }
```

```
};
class C : public A //C is also derived from class base
{
   public:
     void sum()
     {
              cout << "\nSum= " << x + y;
     }
};
int main()
{
   B obj1;        //object of derived class B
   C obj2;        //object of derived class C
   obj1.getdata();
   obj1.product();
   obj2.getdata();
   obj2.sum();
   return 0;
}  //end of program
```

Program-8:

```
#include <iostream>
using namespace std;
class Shape                // Declaration of base class.
{
   public:
   int a;
   int b;
   void get_data(int n,int m)
   {
      a= n;
      b = m;
   }
};
class Rectangle : public Shape  // inheriting Shape class
{
   public:
   int rect_area()
   {
      int result = a*b;
      return result;
   }
};
class Triangle : public Shape    // inheriting Shape class
{
```

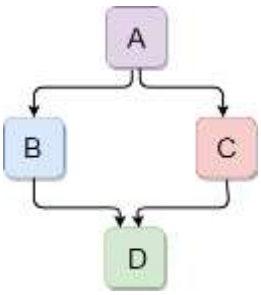
```
    public:
    int triangle_area()
    {
       float result = 0.5*a*b;
       return result;
    }
};
int main()
{
    Rectangle r;
    Triangle t;
    int length,breadth,base,height;
    std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
    cin>>length>>breadth;
    r.get_data(length,breadth);
    int m = r.rect_area();
    std::cout << "Area of the rectangle is : " <<m<< std::endl;
    std::cout << "Enter the base and height of the triangle: " << std::endl;
    cin>>base>>height;
    t.get_data(base,height);
    float n = t.triangle_area();
    std::cout <<"Area of the triangle is : "  << n<<std::endl;
    return 0;
}
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

Hybrid Inheritance:

Hybrid inheritance is a combination of more than one type of inheritance.



Program-9:

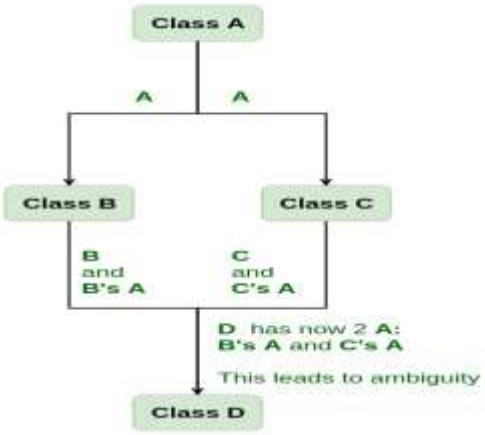
```
#include <iostream>
using namespace std;
class A
{
   protected:
   int a;
   public:
   void get_a()
   {
     std::cout << "Enter the value of 'a' : " << std::endl;
     cin>>a;
   }
};
```

```
class B : public A
{
   protected:
   int b;
   public:
   void get_b()
   {
      std::cout << "Enter the value of 'b' : " << std::endl;
     cin>>b;
   }
};
class C
{
   protected:
   int c;
   public:
   void get_c()
   {
      std::cout << "Enter the value of c is : " << std::endl;
      cin>>c;
   }
};

class D : public B, public C
{
   protected:
   int d;
   public:
   void mul()
   {
       get_a();
       get_b();
       get_c();
       std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
   }
};
int main()
{
   D d;
   d.mul();
   return 0;
}
```

Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

Program-10: (Error)




```
#include<iostream>
using namespace std;
class student {
   int rno;
public:
   void getnumber() {
      cout << "Enter Roll No:";
      cin>>rno;
   }
   void putnumber() {
      cout << "\n\n\tRoll No:" << rno << "\n";
   }
};
class sports : public student {
public:
   int score;
   void getscore() {
      cout << "Enter Sports Score:";
      cin>>score;
   }
   void putscore() {
      cout << "\n\tSports Score is:" << score;
   }
};
class test : public student {
public:
   int part1, part2;
   void getmarks() {
      cout << "Enter Marks\n";
      cout << "Part1:";
      cin>>part1;
      cout << "Part2:";
      cin>>part2;
   }
   void putmarks() {
      cout << "\tMarks Obtained\n";
      cout << "\n\tPart1:" << part1;
      cout << "\n\tPart2:" << part2;
   }
};
class result : public test, public sports {
   int total;
public:
   void display() {
      total = part1 + part2 + score;
```

```
        putnumber();//error
        putmarks();
        putscore();
        cout << "\n\tTotal Score:" << total;
    }
};
int main() {
    result obj;
    obj.getnumber();//error
    obj.getmarks();
    obj.getscore();
    obj.display();
}
```

Program-11: (Virtual)

```
#include<iostream>
using namespace std;
class student {
    int rno;
public:
    void getnumber() {
        cout << "Enter Roll No:";
        cin>>rno;
    }
    void putnumber() {
        cout << "\n\n\tRoll No:" << rno << "\n";
    }
};
class sports : public virtual student {
public:
    int score;
    void getscore() {
        cout << "Enter Sports Score:";
        cin>>score;
    }
    void putscore() {
        cout << "\n\tSports Score is:" << score;
    }
};
class test : virtual public student {
public:
    int part1, part2;
    void getmarks() {
        cout << "Enter Marks\n";
        cout << "Part1:";
        cin>>part1;
        cout << "Part2:";
```

```
        cin>>part2;
    }
    void putmarks() {
        cout << "\tMarks Obtained\n";
        cout << "\n\tPart1:" << part1;
        cout << "\n\tPart2:" << part2;
    }
};
class result : public test, public sports {
    int total;
public:
    void display() {
        total = part1 + part2 + score;
        putnumber();
        putmarks();
        putscore();
        cout << "\n\tTotal Score:" << total;
    }
};
int main() {
    result obj;
    obj.getnumber();
    obj.getmarks();
    obj.getscore();
    obj.display();
}
```

Abstract classes:

Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only.

In C++ class is made abstract by declaring at least one of its functions as **pure virtual function**. A pure virtual function is specified by placing "= 0" in its declaration. Its implementation must be provided by derived classes.

Program-12:

```
#include <iostream>
using namespace std;
 class Shape
{
   public:
   virtual void draw()=0;
};
 class Rectangle : Shape
{
   public:
    void draw()
    {
```

```
        cout < <"drawing rectangle..." < <endl;
    }
};
class Circle : Shape
{
    public:
     void draw()
    {
        cout <<"drawing circle..." < <endl;
    }
};
int main( ) {
    Rectangle rec;
    Circle cir;
    rec.draw();
    cir.draw();
   return 0;
}
```

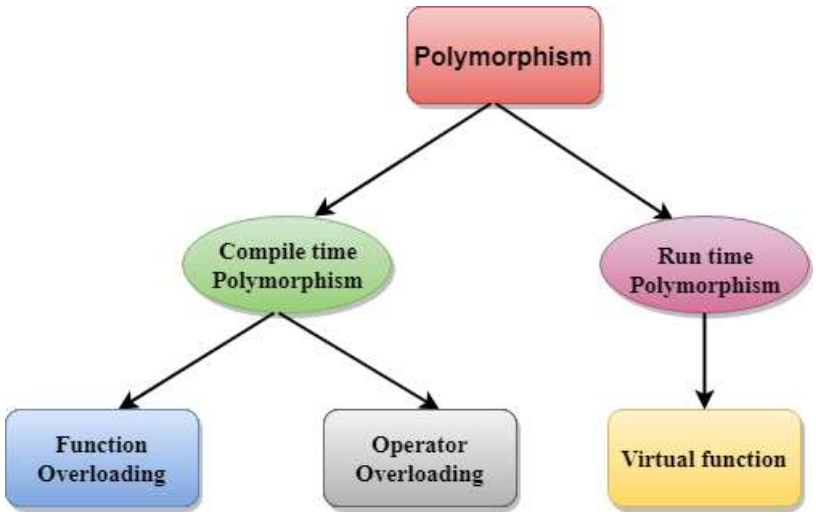
Output:
drawing rectangle...
drawing circle...

Polymorphism:

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. Single name can be used for different purposes.

A real-life example of polymorphism: A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

Different ways of achieving the polymorphism:



Overloading:

- Overloading – A name having two or more distinct meanings
- Overloaded function - a function having more than one distinct meanings
- Overloaded operator - When two or more distinct meanings are defined for an operator

Function Overloading:

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call.

The C++ compiler encodes each function identifier with the number and types of its parameters to enable type-safe linkage.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

Program-13:

```
#include <iostream>
using namespace std;
class Cal {
public:
static int add(int a,int b){
return a + b;
}
static int add(int a, int b, int c)
{
return a + b + c;
}
};
int main(void) {
Cal C;                                  //    class object declaration.
cout<<C.add(10, 20)<<endl;
cout<<C.add(12, 20, 23);
return 0;
}
```

Program-14:

```
void swap (int *a, int *b) ;
void swap (float *c, float *d) ;
void swap (char *p, char *q) ;
int main ( )
{
    int a = 4, b = 6 ;
```

```
    float c = 16.7, d = -7.89 ;
    char p = 'M' , q = 'n' ;
    swap (&a, &b) ;
    swap (&c, &d) ;
    swap (&p, &q) ;
}
void swap (int *a, int *b)
{ int temp;  temp = *a;  *a = *b;  *b = temp; }
void swap (float *c, float *d)
{ float temp;  temp = *c;  *c = *d;  *d = temp; }
void swap (char *p, char *q)
{ char temp;  temp = *p;  *p = *q;  *q = temp; }
```

Program-15:

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);
int mul(int a,int b)
{
return a*b;
}
float mul(float x, int y)
{
return x*y;
}
int main()
{
int r1 = mul(6,7);
float r2 = mul(0.2,3);
cout << "r1 is : " <<r1<< endl;
cout <<"r2 is : "  <<r2<< endl;
return 0;
}
```

Function Overloading and Ambiguity:

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.

- **Type Conversion:**

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
   std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(float j)
{
   std::cout << "Value of j is : " <<j<< std::endl;
}
int main()
{
   fun(12);
   fun(1.2);
   return 0;
}
```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

- **Function with Default Arguments:**

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
   std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
   std::cout << "Value of a is : " <<a<< std::endl;
   std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
   fun(12);
```

```
    return 0;
}
```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

- **Function with pass by reference:**

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
int a=10;
fun(a); // error, which f()?
return 0;
}
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
}
```

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

Operators Overloading:

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operators that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading:

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

Program-15: Overloading of ++ Operator

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
  public:
     Test(): num(8){}
     void operator ++()        {
       num = num+2;
     }
     void Print() {
        cout<<"The Count is: "<<num;
     }
};
int main()
{
```

```
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Program-16: Overloading of Binary + Operator

```
#include <iostream>
using namespace std;
class A
{
     int x;
     public:
     A(){}
    A(int i)
    {
      x=i;
    }
    void operator+(A);
    void display();
};

void A :: operator+(A a)
{

    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;

}
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

Program-17: Overloading of Binary + Operator

```
class complex{
  int real,imag;
  public:
    complex(){
     real=0; imag=0;
    }
    complex(int x,int y){
```

```
    real=x; imag=y;
   }
   void disp(){
    cout<<"\nreal value="<<real<<endl;
    cout<<"imag value="<<imag<<endl;
   }
   complex operator + (complex);
};
complex complex::operator + (complex c){
  complex tmp;
  tmp.real = real + c.real;
  tmp.imag = imag + c.imag;
  return tmp;
}
int main()
{
  complex c1(4,6),c2(7,9);
  complex c3;
  c3 = c1 + c2;
  c1.disp();
  c2.disp();
  c3.disp();
  return 0;
}
```

Program-18: Overloading of Unary - Operator

```
class space {
  int x,y,z;
  public:
  space(){
    x=y=z=0;}
  space(int a, int b,int c){
    x=a; y=b; z=c; }
  void display(){
   cout<<"\nx="<<x<<",y="<<y<<",z="<<z;
  }
  void operator-();
};
void space::operator-() {
  x=-x;
  y=-y;
  z=-z;
}
int main()
{
  space s1(5,4,3);
```

```
s1.display();
-s1;
s1.display();
return 0;
}
```

Program-18: Overloading Prefix and Postfix operator(++)

```
class demo
{
   int m;
   public:
    demo(){ m = 0;}
    demo(int x)
    {
      m = x;
    }
    void operator ++()
    {
       ++m;
       cout<<"Pre Increment="<<m;
    }
    void operator ++(int)
    {
       m++;
       cout<<"Post Increment="<<m;
    }
};
int main()
{
   demo d1(5);
   ++d1;
   d1++;
}
```

Function Overriding:

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

```
#include <iostream>
using namespace std;
class Animal {
   public:
void eat(){
cout<<"Eating...";
   }
```

```
};
class Dog: public Animal
{
 public:
 void eat()
   {
     cout<<"Eating bread...";
   }
};
int main(void) {
  Dog d = Dog();
  d.eat();
  return 0;
}
```

Output:

Eating bread...

Static vs. Dynamic Binding:

Static Binding: the determination of which method to call at compile time

- *Also known as Early Binding*
- Allows greater execution speed
- Achieved through optimized code

Dynamic Binding: the determination of which method to call at run time

- *Also known Late Binding*
- Allows for greater flexibility
- Opportunity for abstraction

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

UNIT -3

Virtual Functions and Pure Virtual Functions, Templates – Class Templates, Function Templates, Templates with Multiple Parameters, Member Function Templates, Overloading of Template Functions, Exception Handling – Exception Handling Mechanism, Throwing Mechanism, Catching Mechanism, Re-throwing an Exceptions and Specifying Exceptions.

Virtual Function:

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

Program-1:

```
#include <iostream>
using namespace std;
class A
```

```
{
  int x=5;
   public:
   void display()
   {
      std::cout << "Value of x is : " << x<<std::endl;
   }
};
class B: public A
{
   int y = 10;
   public:
   void display()
   {
      std::cout << "Value of y is : " <<y<< std::endl;
   }
};
int main()
{
   A *a;
   B b;
   a = &b;
  a->display();
   return 0;
}
```

Output:

```
Value of x is : 5
```

In the above example, * a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

Program-2: Virtual function Example:

```
#include <iostream>
using namespace std;
class A
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
```

```
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
 B b;     //object of derived class
 a = &b;
 a->display();   //Late Binding occurs
}
```

Output:

```
Derived Class is invoked
```

Program-3:

```
class A {
public:
  void x() {cout<<"A::x";};
  virtual void y() {cout<<"A::y";};
};
class B : public A {
public:
  void x() {cout<<"B::x";};
  virtual void y() {cout<<"B::y";};
};
int main () {
   B b;
   A *ap = &b; B *bp = &b;
   b.x (); // prints "B::x"
   b.y (); // prints "B::y"
   bp->x (); // prints "B::x"
   bp->y (); // prints "B::y"
   ap->x (); // prints "A::x"
   ap->y (); // prints "B::y"
   return 0;
};
```

Program-4: Virtual Destructors--example

```
class A {
public:
  A () {cout<<" A";}
  virtual ~A () {cout<<" ~A";}
  virtual f(int);
};
class B : public A {
public:
  B () :A() {cout<<" B";}
  virtual ~B() {cout<<" ~B";}
  virtual f(int) override; //C++11
};
int main () {
   // prints "A B"
   A *ap = new B;
   // prints "~B ~A" : would only
   // print "~A" if non-virtual
   delete ap;
   return 0;
};
```

Pure Virtual Function:

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

```
virtual void display() = 0;
```

Program-5: Pure virtual function

```
#include <iostream>
using namespace std;
class Base
{
   public:
```

```
    virtual void show() = 0;
};
class Derived : public Base
{
    public:
    void show()
    {
        std::cout << "Derived class is derived from the base class." << std::endl;
    }
};
int main()
{
    Base *bptr;
    //Base b;
    Derived d;
    bptr = &d;
    bptr->show();
    return 0;
}
```

Output:

```
Derived class is derived from the base class.
```

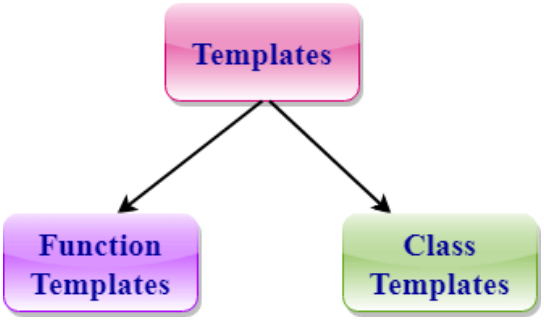
In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

C++ Templates:

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Templates can be represented in two ways:

- Function templates
- Class templates



Function Templates:
We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

Class Template:
We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

Function Template:

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Syntax of Function Template

```
template < class Ttype> ret_type func_name(parameter_list)
{
   // body of function.
}
```

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

Program-6: Function Templates:

```
#include <iostream>
using namespace std;
template<class T> T add(T &a,T &b)
{
   T result = a+b;
   return result;

}
int main()
{
 int i =2;
 int j =3;
 float m = 2.3;
 float n = 1.2;
 cout<<"Addition of i and j is :"<<add(i,j);
 cout<<'\n';
 cout<<"Addition of m and n is :"<<add(m,n);
 return 0;
}
```

Output:

```
Addition of i and j is :5
Addition of m and n is :3.5
```

Function Templates with Multiple Parameters:

We can use more than one generic type in the template function by using the comma to separate the list.

Syntax

```
template<class T1, class T2,.....>
return_type function_name (arguments of type T1, T2....)
{
   // body of function.
}
```

Program-7:

```
#include <iostream>
using namespace std;
template<class X,class Y> void fun(X a,Y b)
{
   std::cout << "Value of a is : " <<a<< std::endl;
   std::cout << "Value of b is : " <<b<< std::endl;
}

int main()
{
  fun(15,12.3);

  return 0;
}
```

Output:

```
Value of a is : 15
Value of b is : 12.3
```

Overloading a Function Template

We can overload the generic function means that the overloaded template functions can differ in the parameter list.

Program-8:

```
#include <iostream>
using namespace std;
template<class X> void fun(X a)
{
   std::cout << "Value of a is : " <<a<< std::endl;
}
```

```
template<class X,class Y> void fun(X b ,Y c)
{
   std::cout << "Value of b is : " <<b<< std::endl;
   std::cout << "Value of c is : " <<c<< std::endl;
}
int main()
{
  fun(10);
  fun(20,30.5);
  return 0;
}
```

Output:

```
Value of a is : 10
Value of b is : 20
Value of c is : 30.5
```

Restrictions of Generic Functions

Generic functions perform the same operation for all the versions of a function except the data type differs.

Program-9:

```
#include <iostream>
using namespace std;
void fun(double a)
{
   cout<<"value of a is : "<<a<<'\n';
}

void fun(int b)
{
   if(b%2==0)
   {
      cout<<"Number is even";
   }
   else
   {
      cout<<"Number is odd";
   }

}

int main()
{
  fun(4.6);
```

```
    fun(6);
    return 0;
}
```

Output:

```
value of a is : 4.6
Number is even
```

In the above example, we overload the ordinary functions. We cannot overload the generic functions as both the functions have different functionalities. First one is displaying the value and the second one determines whether the number is even or not.

CLASS TEMPLATE:

Class Template can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

Syntax

```
template<class Ttype>
class class_name
{
  .
  .
}
```

Ttype is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

Now, we create an instance of a class

```
class_name<type> ob;
```

where class_name: It is the name of the class.
type: It is the type of the data that the class is operating on.
ob: It is the name of the object.

Program-10:

```
#include <iostream>
using namespace std;
template<class T>
class A
{
   public:
   T num1 = 5;
```

```
    T num2 = 6;
    void add()
    {
        std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
    }

};

int main()
{
    A<int> d;
    d.add();
    return 0;
}
```

Output:

```
Addition of num1 and num2 : 11
```

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, .....>
class class_name
{
  // Body of the class.
}
```

Program-11:

```
#include <iostream>
    using namespace std;
    template<class T1, class T2>
   class A
   {
       T1 a;
       T2 b;
       public:
      A(T1 x,T2 y)
     {
```

```
        a = x;
        b = y;
    }
        void display()
        {
            std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;
        }
};

int main()
{
    A<int,float> d(5,6.5);
    d.display();
    return 0;
}
```

Output:

Values of a and b are : 5,6.5

Nontype Template Arguments:

The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types.

```
template<class T, int size>
class array
{
    T arr[size];        // automatic array initialization.
};
```

In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

Arguments are specified when the objects of a class are created:

```
array<int, 15> t1;          // array of 15 integers.
array<float, 10> t2;        // array of 10 floats.
array<char, 4> t3;          // array of 4 chars.
```

Program-12:

```
#include <iostream>
using namespace std;
template<class T, int size>
class A
{
  public:
```

```
    T arr[size];
    void insert()
    {
       int i =1;
       for (int j=0;j<size;j++)
       {
          arr[j] = i;
          i++;
       }
    }

    void display()
    {
       for(int i=0;i<size;i++)
       {
          std::cout << arr[i] << " ";
       }
    }
};
int main()
{
    A<int,10> t1;
    t1.insert();
    t1.display();
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

In the above example, the class template is created which contains the nontype template argument, i.e., size. It is specified when the object of class 'A' is created.

Points to Remember

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use nontype arguments such as built-in or derived data types as template arguments.

Exception Handling:

Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.
In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Advantage:
It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

C++ Exception Handling Keywords:

In C++, we use 3 keywords to perform exception handling:

- try
- catch, and
- throw

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw** − A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try** − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows −

```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using **throw** statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs −

```
double division(int a, int b) {
  if( b == 0 ) {
    throw "Division by zero condition!";
  }
  return (a/b);
}
```

Catching Exceptions

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
  // protected code
} catch( ExceptionName e ) {
 // code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows −

```
try {
  // protected code
} catch(...) {
 // code to handle any exception
}
```

Program-13: Example without Try/Catch

```
#include <iostream>
using namespace std;
float division(int x, int y) {
  return (x/y);
}
int main () {
  int i = 50;
  int j = 0;
```

```
  float k = 0;
    k = division(i, j);
    cout << k << endl;
  return 0;
}
```

Output:

```
Floating point exception (core dumped)
```

Program-14: Example with Try/Catch

```
#include <iostream>
using namespace std;
float division(int x, int y) {
  if( y == 0 ) {
    throw "Attempted to divide by zero!";
  }
  return (x/y);
}
int main () {
  int i = 25;
  int j = 0;
  float k = 0;
  try {
    k = division(i, j);
    cout << k << endl;
  }catch (const char* e) {
    cerr << e << endl;
  }
  return 0;
}
```

Output:

```
Attempted to divide by zero!
```

Re-throwing an Exception:

```
/*## Simple C++ Program for Rethrowing Exception Handling in Function*/
/*## Exception Handling C++ Programs, Exception Handling Programming*/
// Header Files
#include <iostream>
using namespace std;
// Function for Exception Thrown
void exceptionFunction() {
```

```
    // try block - inside Function
    try {
      //throw exception : In function
      throw 0;
    } catch (int i) {
      cout << "\nIn Function : Wrong Input :" << i;
      //re throw exception : out of the function
      throw;
    }
}

int main() {
    int var = 0;

    cout << "Simple C++ Program for Rethrowing Exception Handling : In Function\n";
    // try block - for exception code
    try {
      // Inside try block
      exceptionFunction();
    }// catch block
    catch (int ex) {
      // Code executed when exception caught
      cout << "\nIn Main : Wrong Input :" << ex;
    }

    return 0;
}
```

RAJEEV GANDHI MEMORIAL COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous)

Lecture Notes
Programming in C++ and Data Structures unit-4

UNIT-IV

A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. Below is an overview of some popular linear data structures.

1. **Array**
2. **Linked List**
3. **Stack**
4. **Queue**

Array

Array is a data structure used to store homogeneous elements at contiguous locations. Size of an array must be provided before storing data.

Let size of array be n.

Accessing Time: O(1) [This is possible because elements are stored at contiguous locations] Search Time: O(n) for Sequential Search: O(log n) for Binary Search [If Array is sorted]

Insertion Time: O(n) [The worst case occurs when insertion happens at the Beginning of an array and requires shifting all of the elements]

Deletion Time: O(n) [The worst case occurs when deletion happens at the Beginning of an array and requires shifting all of the elements]

Example : For example, let us say, we want to store marks of all students in a class, we can use an array to store them. This helps in reducing the use of number of variables as we don't need to create a separate variable for marks of every subject. All marks can be accessed by simply traversing the array.

Linked List

A linked list is a linear data structure (like arrays) where each element is a separate object. Each element (that is node) of a list is comprising of two items – the data and a reference to the next node.

Types of Linked List :

1. **Singly Linked List :** In this type of linked list, every node stores address or reference of next node in list and the last node has next address or reference as NULL. For example 1->2->3->4->NULL
2. **Doubly Linked List :** In this type of Linked list, there are two references associated with each node, One of the reference points to the next node and one to the previous node. Advantage of this data structure is that we can traverse in both the directions and for deletion we don't need to have explicit access to previous node. Eg. NULL<-1<->2<->3->NULL
3. **Circular Linked List :** Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list. Advantage of this data structure is that any node can be made as starting node. This is useful in implementation of circular queue in linked list. Eg. 1->2->3->1 [The next pointer of last node is pointing to the first]

Accessing time of an element : O(n)

Search time of an element : O(n)

Insertion of an Element : O(1) [If we are at the position where we have to insert an element]

Deletion of an Element : O(1) [If we know address of node previous the node to be deleted]

Example : Consider the previous example where we made an array of marks of student. Now if a new subject is added in the course, its marks also to be added in the array of marks. But the size of the array was fixed and it is already full so it can not add any new element. If we make an array of a size lot more than the number of subjects it is possible that most of the array will remain empty. We reduce the space wastage Linked List is

formed which adds a node only when a new element is introduced. Insertions and deletions also become easier with linked list.

One big drawback of linked list is, random access is not allowed. With arrays, we can access i'th element in O(1) time. In linked list, it takes Θ(i) time.

Stack

A stack or LIFO (last in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the last element that was added. In stack both the operations of push and pop takes place at the same end that is top of the stack. It can be implemented by using both array and linked list.

Insertion : O(1)

Deletion : O(1)

Access Time : O(n) [Worst Case]

Insertion and Deletion are allowed on one end.

Example : Stacks are used for maintaining function calls (the last called function must finish execution first), we can always remove recursion with the help of stacks. Stacks are also used in cases where we have to reverse a word, check for balanced parenthesis and in editors where the word you typed the last is the first to be removed when you use undo operation. Similarly, to implement back functionality in web browsers.

Queue

A queue or FIFO (first in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: enqueue, the process of adding an element to the collection.(The element is added from the rear side) and dequeue, the process of removing the first element that was added. (The element is removed from the front side). It can be implemented by using both array and linked list.

Insertion : O(1)

Deletion : O(1)

Access Time : O(n) [Worst Case]

Example : Queue as the name says is the data structure built according to the queues of bus stop or train where the person who is standing in the front of the queue(standing for the longest time) is the first one to get the ticket. So any situation where resources are shared among multiple users and served on first come first server basis. Examples include CPU scheduling, Disk Scheduling. Another application of queue is when data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

Circular Queue

The advantage of this data structure is that it reduces wastage of space in case of array implementation, As the insertion of the (n+1)'th element is done at the 0'th index if it is empty.

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a

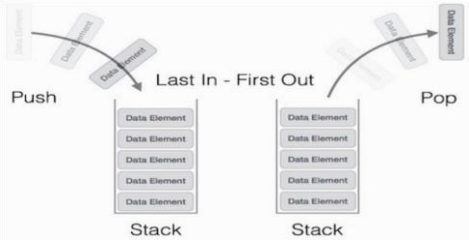
deck of cards or a pile of plates, etc. 

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

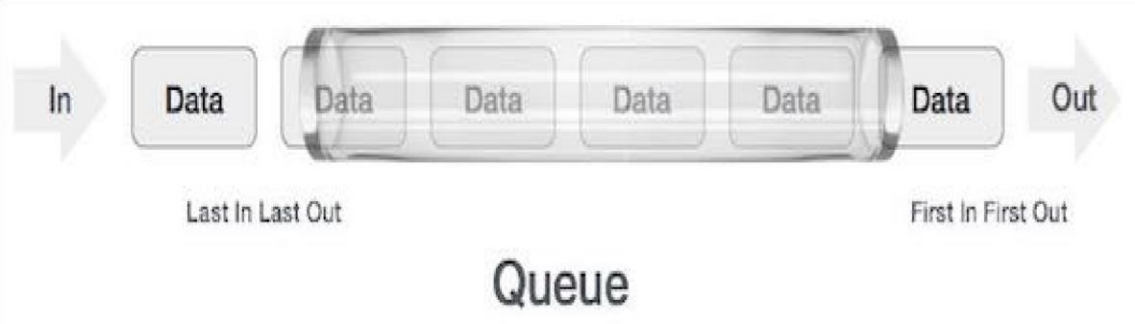
- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

QUEUE: Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-InFirst-Out methodology, i.e., the data item stored first will be accessed first.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using onedimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.

- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer. Let's first learn about supportive functions of a queue − peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek
  return queue[front]
end procedure
```

Implementation of peek() function in C programming language − **Example**

```
int     peek()     {
return
queue[front];

}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull
  if rear equals to MAXSIZE
        return true
else
        return false
endif
end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {    if(rear
==  MAXSIZE  -  1)
return  true;        else
return false;

}
```

isempty()

Algorithm of isempty() function − **Algorithm**

```
begin procedure isempty
   if front is less than MIN  OR front is greater than rear
return true
 else
return false
 endif
 end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

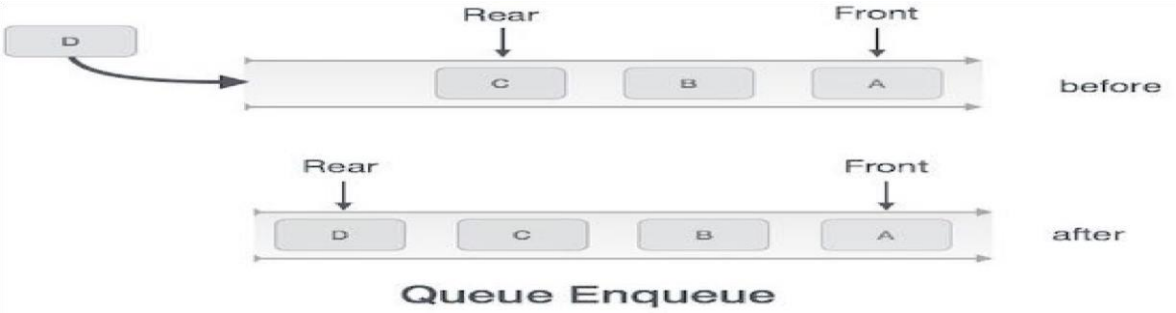
```
bool isempty() {    if(front
<  0  ||  front  >  rear)
return  true;            else
return false;

}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue − **Step 1** − Check if the queue is full.

- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)        if
queue is full        return overflow
endif
    rear ← rear + 1
    queue[rear] ← data
    return true
  end procedure
```

Implementation of enqueue() in C programming language − **Example**

```
int enqueue(int data)

  if(isfull())
return 0;

   rear = rear + 1;    queue[rear] = data;

   return 1;

end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.

Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue
if queue is empty
 return underflow
 end if
```

```
  data = queue[front]
front ← front + 1
    return true end procedure
```

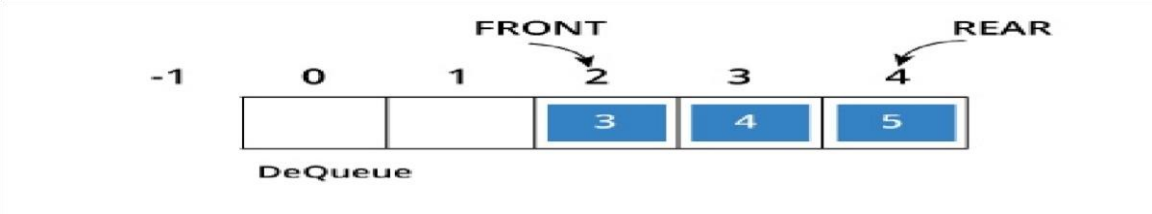
Implementation of dequeue() in C programming language − **Example**

```
int dequeue() {


  if(isempty())
return 0;      int
data              =
queue[front];
front = front +
1;


  return data;

}
```

LIST: In computer science, a **list** or **sequence** is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a finite sequence; the (potentially) infinite analog of a list is a stream.[1]:§3.5 Lists are a basic example of containers, as they contain other values. If the same value occurs multiple times, each occurrence is considered a distinct item.

Many programming languages provide support for **list data types**, and have special syntax and semantics for lists and list operations. A list can often be constructed by writing the items in sequence, separated by commas, semicolons, or spaces

Circular Queue: Circular queue avoids the wastage of space in a regular queue implementation using arrays.



Circular Queue

As you can see in the above image, after a bit of enqueueing and dequeueing, the size of the queue has been reduced.

The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued. **How Circular Queue Works**

Circular Queue works by the process of circular increment i.e. when we try to increment any variable and we reach the end of queue, we start from the beginning of queue by modulo division with the queue size. i.e.

if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)

Queue operations work as follows:

- Two pointers called *FRONT* and *REAR* are used to keep track of the first and last elements in the queue.
- When initializing the queue, we set the value of *FRONT* and *REAR* to -1.
- On enqueing an element, we circularly increase the value of *REAR* index and place the new element in the position pointed to by *REAR*.
- On dequeueing an element, we return the value pointed to by *FRONT* and circularly increase the *FRONT* index.
- Before enqueing, we check if queue is already full.
- Before dequeuing, we check if queue is already empty.
- When enqueing the first element, we set the value of *FRONT* to 0.
- When dequeing the last element, we reset the values of *FRONT* and *REAR* to -1.

However, the check for full queue has a new additional case:

- Case 1: *FRONT* = 0 & & REAR == SIZE - 1
- Case 2: FRONT = REAR + 1

The second case happens when *REAR* starts from 0 due to circular increment and when its value is just 1 less than *FRONT*, the queue is full.

Circular Queue Implementation in programming language

The most common queue implementation is using arrays, but it can also be implemented using lists.

Implementation using C++ programming

```
#include <iostream> #define SIZE 5
/* Size of Circular Queue */
using namespace std;
class Queue {
private:
int items[SIZE], front,
rear;
    public:     Queue(){
 front = -1;
rear = -1;
}
bool isFull(){
        if(front == 0 && rear ==
SIZE - 1){             return true;
}
        if(front == rear + 1) {
 return true;          }
 return false;     }
        bool isEmpty(){
        if(front == -1)
return true;
else
return false;
```

```
    }
        void
enQueue(int   element){
if(isFull()){
            cout << "Queue is
full";         }
else {
            if(front == -1)
front = 0;
rear = (rear + 1) % SIZE;
items[rear] = element;
cout << endl << "Inserted " << element << endl;
}
    }
        int deQueue(){
int element;
if(isEmpty()){
cout << "Queue is empty" << endl;
return(-1);
}
else
{
 element = items[front];
if(front == rear){
front = -1;
 rear = -1;
            } /* Q has only one element, so we reset the
queue after deleting it. */
else {
                front=(front+1)
% SIZE;             }
```

```
return(element);
}
  void display()
    {
        /* Function to display status of Circular
Queue */
int i;
if(isEmpty()) {
            cout << endl << "Empty Queue"
<< endl;         }         else         {
            cout << "Front -> " << front;              cout
<< endl << "Items -> ";
 for(i=front; i!=rear;i=(i+1)%SIZE)
 cout << items[i];
cout << items[i];
cout << endl << "Rear -> " << rear;         }
    }
    };
int main() {
    Queue q;
        // Fails because
front = -1
    q.deQueue();

    q.enQueue(1);
    q.enQueue(2);
    q.enQueue(3);
    q.enQueue(4);
    q.enQueue(5);
```

```
    // Fails to enqueue because front == 0 && rear == SIZE -
1
    q.enQueue(6);

    q.display();
int elem = q.deQueue();
if( elem != -1)
       cout << endl << "Deleted Element is " << elem;
    q.display();
    q.enQueue(7);
    q.display();
        // Fails to enqueue because front
== rear + 1

q.enQueue(8);
return 0; }
When you run this program, the output will be
Queue is empty
Inserted 1
Inserted 2
Inserted 3
Inserted 4
```

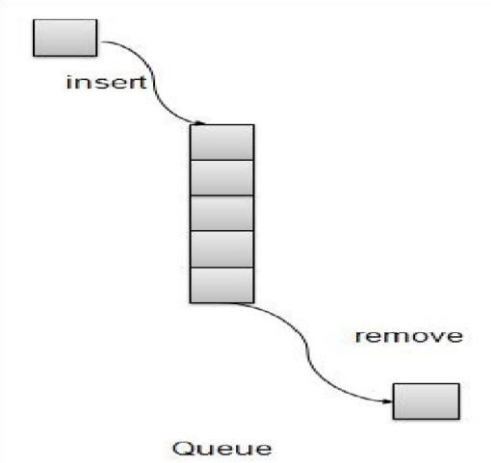
Priority Queue:

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

- **insert / enqueue** − add an item to the rear of the queue.
- **remove / dequeue** − remove an item from the front of the queue.

Priority Queue Representation

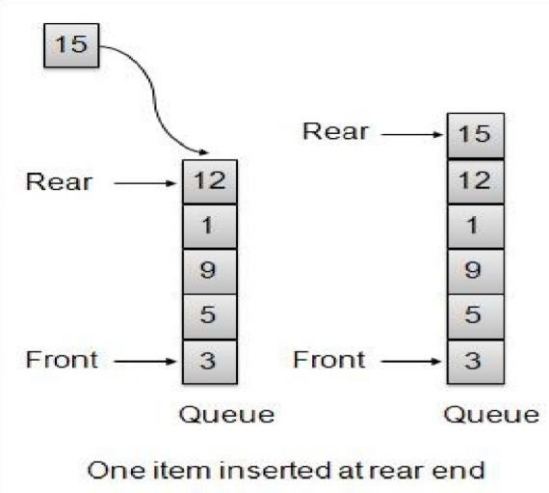


We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- **Peek** − get the element at front of the queue.
- **isFull** − check if queue is full.
- **isEmpty** − check if queue is empty.

Insert / Enqueue Operation

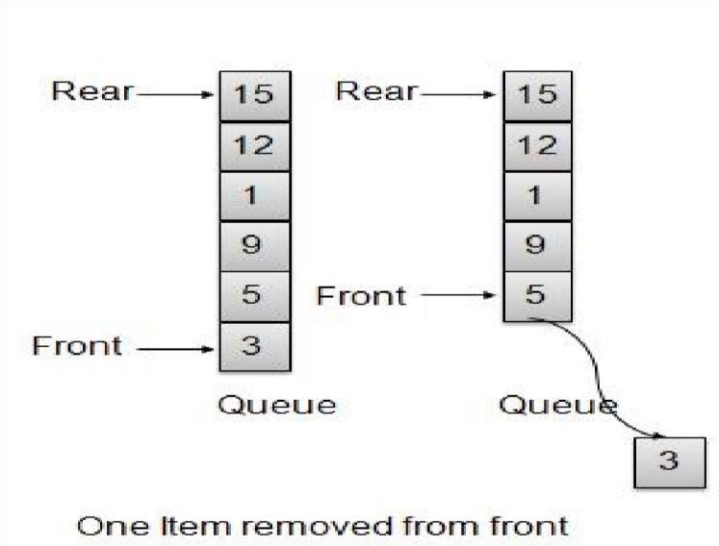
Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.




```
void insert(int data){
  int i = 0;
  if(!isFull()){
    // if queue is empty, insert the data
          if(itemCount == 0){          intArray[itemCount++] = data;
    }else{
      // start from the right end of the queue
for(i = itemCount - 1; i >= 0; i-- ){
        // if data is larger, shift existing item to right end
if(data > intArray[i]){            intArray[i+1] = intArray[i];
        }else{
break;
        }
      }
      // insert the data
intArray[i+1] = data;
itemCount++;

    }

  }

}
```

Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.



One Item removed from front

```
int removeData(){      return
intArray[--itemCount];
}
```

Demo Program

PriorityQueueDemo.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include          <stdbool.h>
#define MAX 6
int
intArray[MAX]
; int itemCount
= 0;
int    peek(){                return
intArray[itemCount - 1];
}
```

```
bool isEmpty(){
return itemCount ==
0;
}
bool isFull(){ return
itemCount == MAX;
}
int size(){
return
itemCount; }
void insert(int
data){ int i = 0;
if(!isFull()){
  // if queue is empty, insert the data
if(itemCount == 0){
intArray[itemCount++] = data;
  }else{
    // start from the right end of the queue

    for(i = itemCount - 1; i >= 0; i-- ){
      // if data is larger, shift existing item to right end
if(data > intArray[i]){ intArray[i+1] = intArray[i];
      }else{
break;
      }
    }
```

```
        //      insert      the      data
intArray[i+1]          =          data;
itemCount++;
     }
  }
}
int removeData(){      return
intArray[--itemCount];
}
int main() {    /*
insert 5 items */
insert(3);
insert(5);
  insert(9);
insert(1);
insert(12);
  // ----------------
-   // index : 0  1
2 3 4
  // ------------------
  //   queue   :   12   9   5   3   1
insert(15);
  //  -------------------
-   // index : 0  1 2 3
4  5
  // ---------------------
  // queue : 15 12 9 5 3 1
        if(isFull()){
printf("Queue  is  full!\n");
}
```

```
  // remove one item       int num =
removeData();             printf("Element
removed: %d\n",num);

  // -----------------
---  // index : 0  1
2 3 4
  //  --------------------
// queue : 15 12 9 5 3
  //     insert     more     items
insert(16);
  // ----------------------
  // index :  0  1 2 3 4  5
  // ----------------------
  // queue : 16 15 12 9 5 3
  // As queue is full, elements will not be inserted.
  insert(17);
insert(18);
  // ----------------------
  // index : 0   1  2 3 4 5
  // ----------------------
  //   queue   :   16   15   12   9   5   3
printf("Element at front: %d\n",peek());
  printf("----------------------
\n");     printf("index : 5 4 3 2
1  0\n");    printf("--------------
--------\n");       printf("Queue:
");

  while(!isEmpty()){       int n
=                  removeData();
printf("%d ",n);
```

```
   }
}
```

If we compile and run the above program then it would produce following result − Queue is full!

Element removed: 1

Element at front: 3

---

index : 5 4 3 2 1 0

Queue: 3 5 9 12 15 16

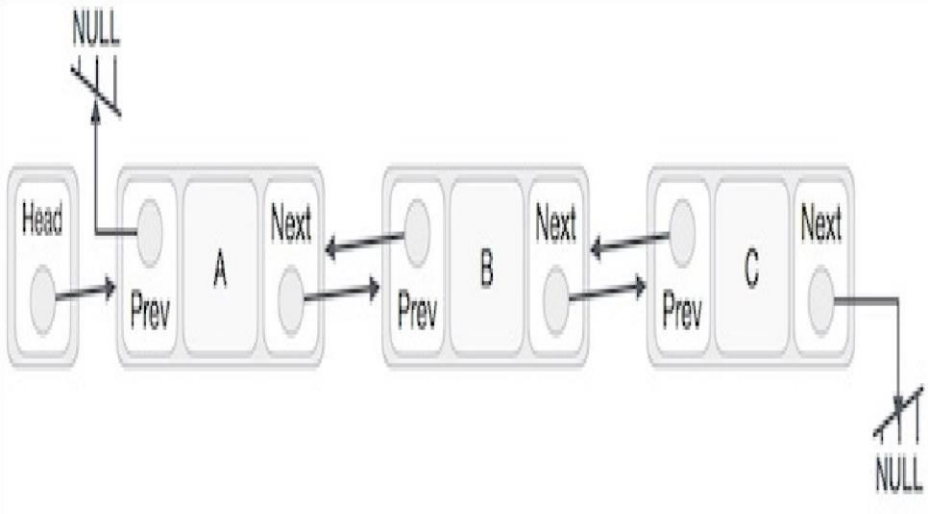
Double Linked List:

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.

Following are the important terms to understand the concept of doubly linked list.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **Prev** − Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Insert Last** − Adds an element at the end of the list.
- **Delete Last** − Deletes an element from the end of the list.
- **Insert After** − Adds an element after an item of the list.
- **Delete** − Deletes an element from the list using the key.
- **Display forward** − Displays the complete list in a forward manner.

- **Display backward** − Displays the complete list in a backward manner.

Insertion Operation

Following code demonstrates the insertion operation at the beginning of a doubly linked list.

Example

```
//insert link at the first location
void insertFirst(int key, int data)
{
  //create a link          struct node *link = (struct node*)
malloc(sizeof(struct node));   link->key = key;   link->data = data;

  if(isEmpty()) {
    //make it the last link
last = link;
  } else {
    //update first prev
link       head->prev =
link;    }
  //point it to old first link    link-
>next = head;

  //point first to new first link
head = link;
}
```

Deletion Operation

Following code demonstrates the deletion operation at the beginning of a doubly linked list.

Example

```
//delete first item struct
node* deleteFirst() {
```

```
  //save reference to first link
struct node *tempLink = head;

  //if only one link    if(head-
>next == NULL) {       last =
NULL;     } else {       head-
>next->prev = NULL;
  }

  head = head->next;

  //return the deleted
link                return
tempLink; }
```

Insertion at the End of an Operation

Following code demonstrates the insertion operation at the last position of a doubly linked list.

Example

```
//insert link at the last location
void insertLast(int key, int data)
{
  //create a link          struct node *link = (struct node*)
malloc(sizeof(struct node));    link->key = key;
  link->data = data;

  if(isEmpty())       {
//make it the last link
last = link;
  } else {
    //make link a new last link       last-
>next = link;
```

```
      //mark old last node as prev of new link
link->prev = last;    }
  //point last to new last node
last = link;
}
```

RAJEEV GANDHI MEMORIAL COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous)

Lecture Notes
Programming in C++ and Data Structures unit-5

UNIT-V

Binary Search Trees:

A Binary Search Tree (BST) is a tree in which all the nodes follow the belowmentioned properties −

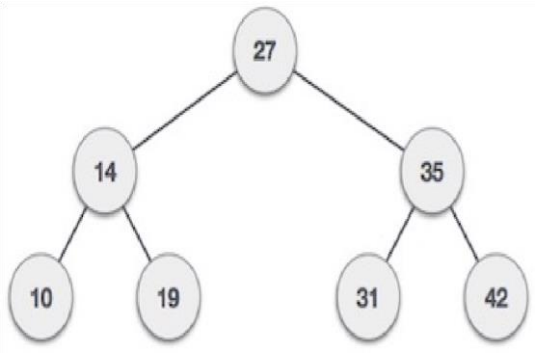
- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved. Following is a pictorial representation of BST −



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

Following are the basic operations of a tree − •

Search − Searches an element in a tree.

- **Insert** − Inserts an element in a tree.
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

Node

Define a node having some data, references to its left and right child nodes.

```
struct node {
  int data;      struct
node *leftChild;
struct node
*rightChild; } ;
```

Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

```
Algorithm struct node*
search(int data){    struct
node *current = root;
printf("Visiting elements:
");


  while(current->data != data){


    if(current != NULL) {
printf("%d ",current->data);
```

```
        //go to left tree
if(current->data > data){
current = current->leftChild;
}//else go to right tree          else {

          current = current->rightChild;

        }


//not found
if(current == NULL){
return NULL;

        }

      }
  }    return
current;

}
```

Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
void insert(int data) {     struct node *tempNode = (struct node*)
malloc(sizeof(struct node));     struct node *current;     struct node
*parent;
```

```
  tempNode->data = data;    tempNode-
>leftChild = NULL;    tempNode->rightChild =
NULL;


  //if tree is empty
if(root == NULL) {
root = tempNode;
  } else {
current =
root;
parent =
NULL;


    while(1) {
parent = current;


       //go to left of the tree         if(data <
parent->data) {            current = current-
>leftChild;
          //insert to the left


          if(current == NULL) {
parent->leftChild = tempNode;
return;
          }
       }//go to right of the tree
else {            current = current-
>rightChild;
```

```
            //insert to the right
if(current == NULL) {              parent-
>rightChild = tempNode;
return;
          }
        }
      }
   }
}
```

or a binary tree to be a binary search tree, the data of all the nodes in the left subtree of the root node should be ≤ the data of the root. The data of all the nodes in the right subtree of the root node should be > the data of the root.

Example

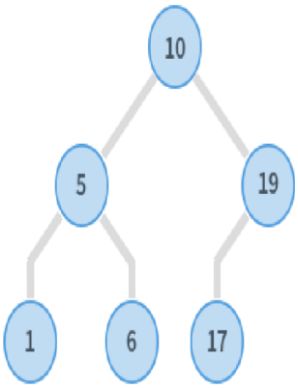


Fig. 1

In Fig. 1, consider the root node with data = 10.

- Data in the left subtree is: [5,1,6]
- All data elements are < 10
- Data in the right subtree is: [19,17]

- All data elements are > 10
-

Also, considering the root node with data=5

, its children also satisfy the specified ordering. Similarly, the root node with data=19

also satisfies this ordering. When recursive, all subtrees satisfy the left and right subtree ordering.

The tree is known as a Binary Search Tree or BST.

Traversing the tree

There are mainly *three* types of tree traversals.

Pre-order traversal

In this traversal technique the traversal order is root-left-right i.e.

- Process data of root node
- First, traverse left subtree completely
- Then, traverse right subtree void perorder(struct node*root) { if(root) {

printf("%d ",root->data); //Printf root->data preorder(root->left); //Go to left subtree preorder(root->right); //Go to right subtree }

}

Post-order traversal

In this traversal technique the traversal order is left-right-root.

- Process data of left subtree
- First, traverse right subtree
- Then, traverse root node void postorder(struct node*root) { if(root) {

postorder(root->left); //Go to left sub tree postorder(root->right); //Go to right sub tree printf("%d ",root->data); //Printf root->data }

}

In-order traversal

In in-order traversal, do the following:

- First process left subtree (before processing root node)
- Then, process current root node
- Process right subtree void inorder(struct node*root) { if(root) {

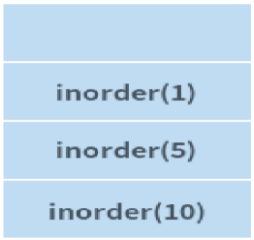
inorder(root->left); //Go to left subtree printf("%d ",root->data); //Printf root->data inorder(root->right); //Go to right subtree }

}

Consider the in-order traversal of a sample BST

- The 'inorder( )' procedure is called with root equal to node with data=10
- Since the node has a left subtree, 'inorder( )' is called with root equal to node with data=5
- Again, the node has a left subtree, so 'inorder( )' is called with root=1
-

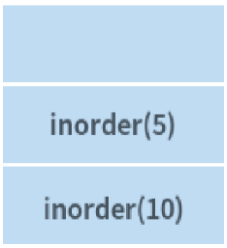
The function call stack is as follows:



- Node with data=1

- does not have a left subtree. Hence, this node is processed.
- Node with data=1
- does not have a right subtree. Hence, nothing is done.
- inorder(1) gets completed and this function call is popped from the call stack.

The stack is as follows:

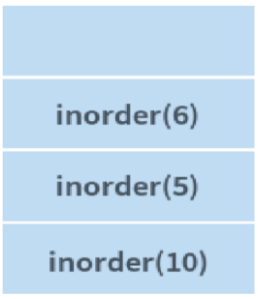


- Left subtree of node with data=5
- is completely processed. Hence, this node gets processed.
- Right subtree of this node with data=5
- is non-empty. Hence, the right subtree gets processed now. 'inorder(6)' is then called.

Note

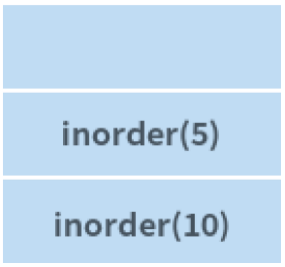
'inorder(6)' is only equivalent to saying inorder(pointer to node with data=6).
The notation has been used for brevity.

The function call stack is as follows:

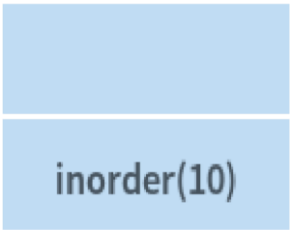


Again, the node with data=6

has no left subtree, Therefore, it can be processed and it also has no right subtree. 'inorder(6)' is then completed.



Both the left and right subtrees of node with data=5 have been completely processed. Hence, inorder(5) is then completed.



- Now, node with data=10
- is processed
- Right subtree of this node gets processed in a similar way as described until step 10
- After right subtree of this node is completely processed, entire traversal of the BST is complete

The order in which BST in Fig. 1 is visited is: 1, 5, 6, 10, 17, 19. The in-order traversal of a BST gives a sorted ordering of the data elements that are present in the BST. This is an important property of a BST.

Insertion in BST

Consider the insertion of data=20
in the BST.

Algorithm

Compare data of the root node and element to be inserted.

1. If the data of the root node is greater, and if a left subtree exists, then repeat step 1 with root = root of left subtree. Else, insert element as left child of current root.
2. If the data of the root node is greater, and if a right subtree exists, then repeat step 2 with root = root of right subtree. Else, insert element as right child of current root.

Implementation

```
struct node* insert(struct node* root, int
data)  {

    if (root == NULL)  //If the tree is empty, return a new,single node

        return newNode(data);      else      {
//Otherwise, recur down the tree          if (data <=
root->data)

            root->left  = insert(root->left, data);          else

            root->right = insert(root->right, data);          //return the
(unchanged) root pointer          return root;      }

  }
```

RAJEEV GANDHI MEMORIAL COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous)

Lecture Notes

Programming in C++ and Data Structures unit-6

UNIT VI

Dictionaries:

Dictionary ADT. Dictionary (map, association list) is a data structure, which is generally an association of unique keys with some values. One may bind a value to a key, delete a key (and naturally an associated value) and lookup for a value by the key. ... In the example, words are keys and explanations are values.

Hash Tables:

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use *hashing*.

In hashing, large keys are converted into small keys by using **hash functions**. The values are then stored in a data structure called **hash table**. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in **O(1)** time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.

2. The element is stored in the hash table where it can be quickly retrieved using hashed key.

hash = hashfunc(key) index =
hash % array_size

In this method, the hash is independent of the array size and it is then reduced to an index (a number between 0 and array_size − 1) by using the modulo operator (%).

Hash function

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

1. Easy to compute: It should be easy to compute and must not become an algorithm in itself.

2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.

3. Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

Note: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Need for a good hash function

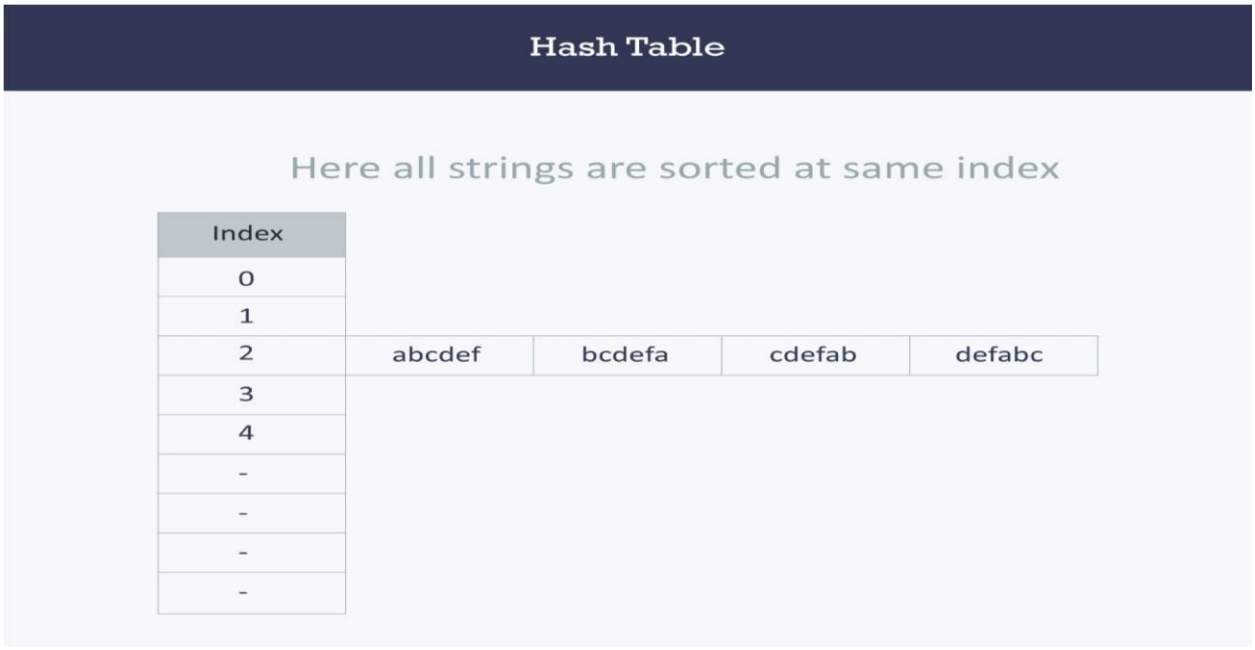
Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.

To compute the index for storing the strings, use a hash function that states the following:

The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.

As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions). It is recommended that you use prime numbers in case of modulo. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively. Since all the strings contain the same characters with different permutations, the sum will 599.

The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format. As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list.



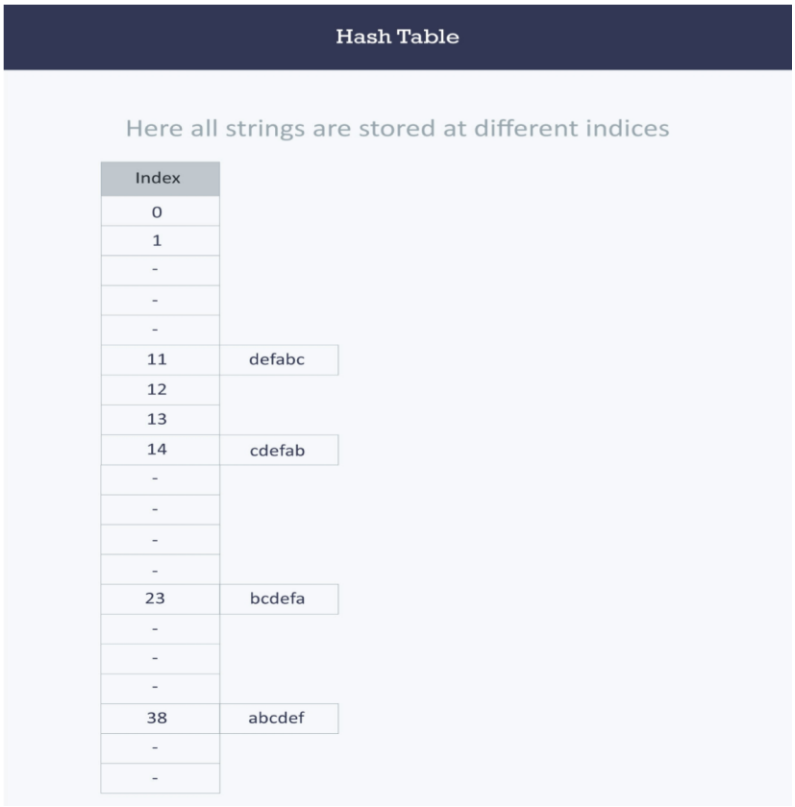
Here, it will take **O(n)** time (where n is the number of strings) to access a specific string. This shows that the hash function is not a good hash function.
Let's try a different hash function. The index for a specific string will be equal to sum of ASCII values of characters multiplied by their respective order in the string after which it is modulo with 2069 (prime number).

Let's try a different hash function. The index for a specific string will be equal to sum of ASCII values of characters multiplied by their respective order in the string after which it is modulo with 2069 (prime number).

String Hash function Index

abcdef (97*1* + *98*2 + 99*3* + *100*4 + 101*5* + *102*6)%2069 38 bcdefa (98*1* + *99*2 + 100*3* + *101*4 + 102*5* + *97*6)%2069 23 cdefab (99*1* + *100*2 + 101*3* +

*102*4 + 97*5* + *986*)%2069 14 defabc (100*1* + *101*2 + 102*3* + *97*4 + 98*5* + *996*)%2069 11



Hash table

A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is **O(1)**.

Let us consider string S. You are required to count the frequency of all the characters in this string. string S = “ababcd”

The simplest way to do this is to iterate over all the possible characters and count their frequency one by one. The time complexity of this approach is **O(26*N)** where **N** is the size of the string and there are 26 possible characters.

```
void countFre(string S)     {

    for(char c = ‘a’;c <= ‘z’;++c)        {
```

```
        int frequency = 0;

        for(int i = 0;i < S.length();++i)                if(S[i] == c)
frequency++;
        cout << c << ' ' << frequency << endl;        }

   }
```

Output

a 2 b 2 c

1 d 1 e 0

f 0 … z 0

Let us apply hashing to this problem. Take an array frequency of size 26 and hash the 26 characters with indices of the array by using the hash function. Then, iterate over the string and increase the value in the frequency at the corresponding index for each character. The complexity of this approach is **O(N)** where **N** is the size of the string.

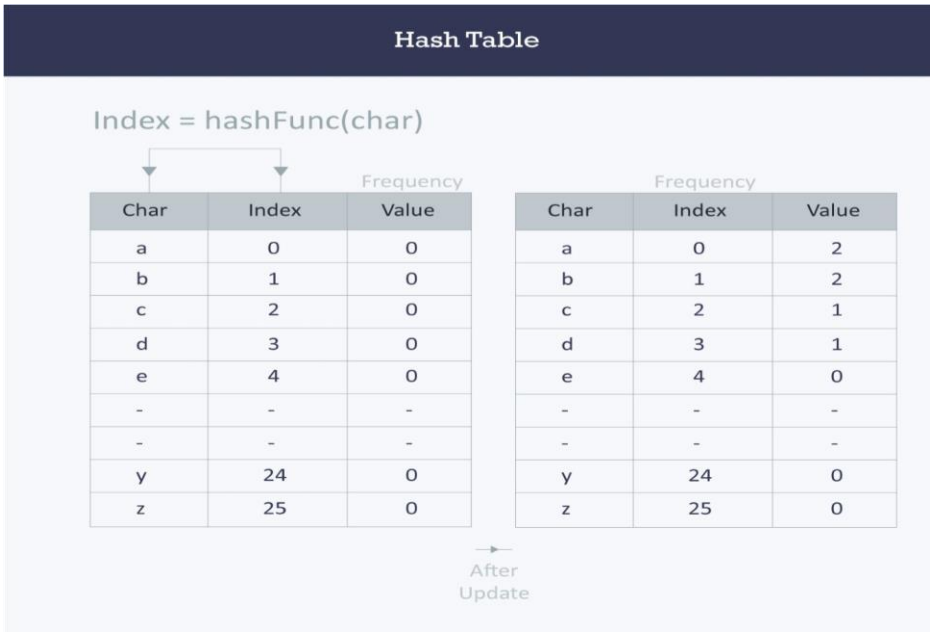
```
 int Frequency[26];


   int hashFunc(char c)    {

      return (c - 'a');    }


   void countFre(string S)    {

      for(int i = 0;i < S.length();++i)        {

         int          index          =          hashFunc(S[i]);
Frequency[index]++;        }

      for(int i = 0;i < 26;++i)

         cout << (char)(i+'a') << ' ' << Frequency[i] << endl;

   }
```

Output

a 2 b 2 c

1 d 1 e 0

f 0 … z 0

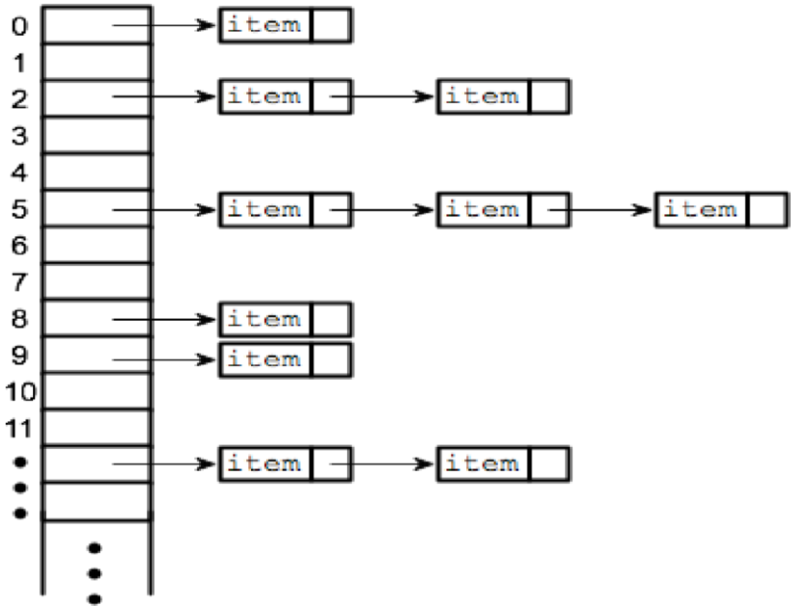


Collision resolution techniques

Separate chaining (open hashing)

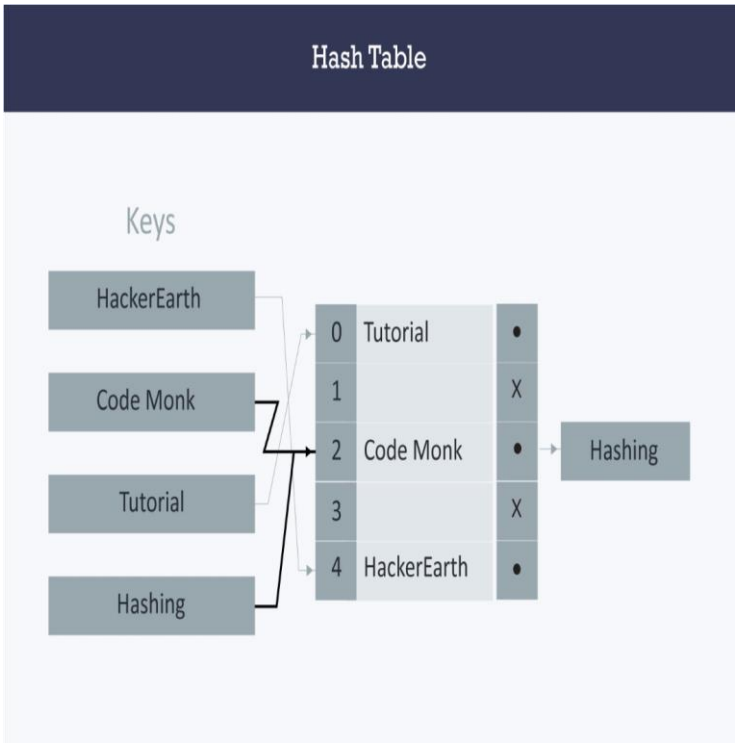
Separate chaining is one of the most commonly used collision resolution techniques. It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.

The cost of a lookup is that of scanning the entries of the selected linked list for the required key. If the distribution of the keys is sufficiently uniform, then the average cost of a lookup depends only on the average number of keys per linked list. For this reason, chained hash tables remain effective even when the number of table entries (N) is much higher than the number of slots.



For separate chaining, the worst-case scenario is when all the entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number (N) of entries in the table.

In the following image, **CodeMonk** and **Hashing** both hash to the value **2**. The linked list at the index **2** can hold only one entry, therefore, the next entry (in this case **Hashing**) is linked (attached) to the entry of **CodeMonk**.



Implementation of hash tables with separate chaining (open hashing) *Assumption*

Hash function will return an integer from 0 to 19.

vector <string> hashTable[20]; int hashTableSize=20;

Insert void insert(string s) { // Compute the index using Hash Function int index = hashFunc(s);

// Insert the element in the linked list at the particular index

hashTable[index].push_back(s); } *Search* void search(string s) { //Compute the index by using the hash function int index = hashFunc(s); //Search the linked list at that specific index for(int i = 0;i < hashTable[index].size();i++) {

if(hashTable[index][i] == s) {

cout << s << " is found!" << endl; return; }

}

```
    cout << s << " is not found!" << endl;     }
```

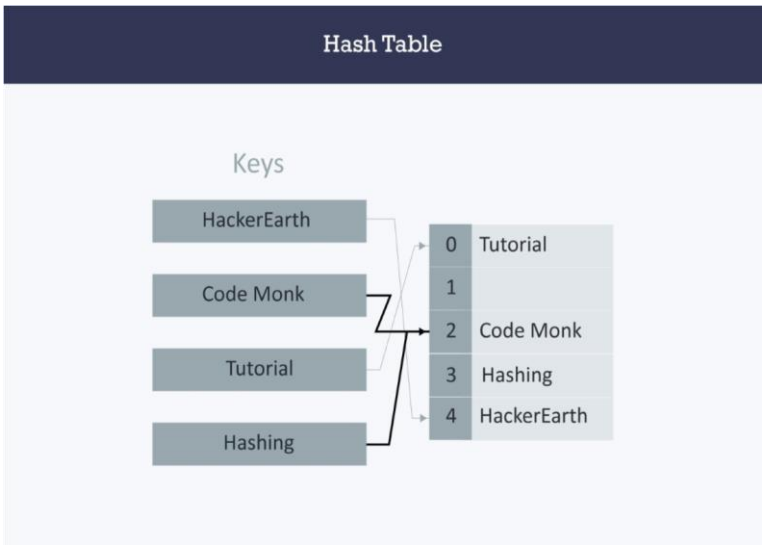
Linear probing (open addressing or closed hashing)

In open addressing, instead of in linked lists, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.
The probe sequence is the sequence that is followed while traversing through entries. In different probe sequences, you can have different intervals between successive entry slots or probes.

When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.

Linear probing is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is **index**. The probing sequence for linear probing will be:

index = index % hashTableSize index =
(index + 1) % hashTableSize index =
(index + 2) % hashTableSize index =
(index + 3) % hashTableSize and so on…



Hash collision is resolved by open addressing with linear probing. Since **CodeMonk** and **Hashing** are hashed to the same index i.e. **2**, store **Hashing** at **3** as the interval between successive probes is **1**.

Implementation of hash table with linear probing

Assumption

- There are no more than 20 elements in the data set.
- Hash function will return an integer from 0 to 19.
- Data set must have unique elements.

```
string hashTable[21];          int
hashTableSize = 21;
```

Insert

```
void insert(string s)    {        //Compute the index using the hash
function        int index = hashFunc(s);

      //Search for an unused slot and if the index will exceed the hashTableSize then roll back
while(hashTable[index] != "")        index = (index + 1) % hashTableSize;      hashTable[index]
= s;    } Search    void search(string s)    {        //Compute the index using the hash function
int index = hashFunc(s);

      //Search for an unused slot and if the index will exceed the hashTableSize then roll back
```

```
while(hashTable[index] != s and hashTable[index] != "")          index = (index +
1) % hashTableSize;              //Check if the element is present in the hash table
if(hashTable[index] == s)          cout << s << " is found!" << endl;        else

        cout << s << " is not found!" << endl;     }
```

Quadratic Probing

Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

Let us assume that the hashed index for an entry is **index** and at **index** there is an occupied slot. The probe sequence will be as follows:

index = index % hashTableSize index = (index + 1^2) % hashTableSize index = (index + 2^2) % hashTableSize index = (index + 3^2) % hashTableSize and so on…

Implementation of hash table with quadratic probing

Assumption

- There are no more than 20 elements in the data set.
- Hash function will return an integer from 0 to 19.
- Data set must have unique elements.

```
string hashTable[21];        int hashTableSize = 21; Insert       void
insert(string s)    {        //Compute the index using the hash function
int index = hashFunc(s);

    //Search for an unused slot and if the index will exceed the hashTableSize roll back        int
h = 1;

    while(hashTable[index] != "")        {
```

```
    index = (index + h*h) % hashTableSize;          h++;    }

  hashTable[index] = s;    } Search void search(string s)    {
//Compute the index using the Hash Function          int index =
hashFunc(s);

   //Search for an unused slot and if the index will exceed the hashTableSize roll back    int
h = 1;

  while(hashTable[index] != s and hashTable[index] != "")
  {

    index = (index + h*h) % hashTableSize;          h++;    }
//Is the element present in the hash table     if(hashTable[index] ==
s)        cout << s << " is found!" << endl;      else

    cout << s << " is not found!" << endl;   }
```

Double hashing

Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions.

Let us say that the hashed index for an entry record is an index that is computed by one hashing function and the slot at that index is already occupied. You must start traversing in a specific probing sequence to look for an unoccupied slot. The probing sequence will be:

index = (index + 1 * indexH) % hashTableSize; index =
(index + 2 * indexH) % hashTableSize; and so on…

Here, **indexH** is the hash value that is computed by another hash function.

Implementation of hash table with double hashing

Assumption

- There are no more than 20 elements in the data set.
- Hash functions will return an integer from 0 to 19.
- Data set must have unique elements.

string hashTable[21]; int hashTableSize = 21; **Insert** void insert(string s) { //Compute the index using the hash function1 int index = hashFunc1(s);

int indexH = hashFunc2(s);

//Search for an unused slot and if the index exceeds the hashTableSize roll back while(hashTable[index] != "")

index = (index + indexH) % hashTableSize; hashTable[index] = s; } **Search** void search(string s) { //Compute the index using the hash function int index = hashFunc1(s); int indexH = hashFunc2(s);

//Search for an unused slot and if the index exceeds the hashTableSize roll back

while(hashTable[index] != s and hashTable[index] != "") index = (index + indexH) % hashTableSize; //Is the element present in the hash table if(hashTable[index] == s) cout << s << " is found!" << endl; else

cout << s << " is not found!" << endl; }

Applications

- *Associative arrays*: Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects).
- *Database indexing*: Hash tables may also be used as disk-based data structures and database indices (such as in dbm).
- *Caches*: Hash tables can be used to implement caches i.e. auxiliary data tables that are used to speed up the access to data, which is primarily stored in slower media.
- *Object representation*: Several dynamic languages, such as Perl, Python, JavaScript, and Ruby use hash tables to implement objects.
- Hash Functions are used in various algorithms to make their computing faster